National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

A Transparent BIST Scheme for Detecting *V*-coupling Faults in RAMs

BY

Nicole Yee-Fen Sat

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

DEPARTMENT OF **ELECTRICAL ENGINEERING**

Edmonton, Alberta

Fall 1995

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-06537-5

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Nicole Yee-Fen Sat**

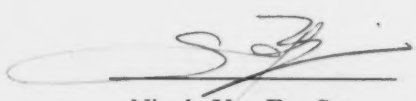
TITLE OF THESIS: **A Transparent BIST Scheme for Detecting *V*-coupling Faults in RAMs**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1995**

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's written permission.



Nicole Yee-Fen Sat
204, Sungai Marong
28700 Bentong
Pahang Darulmakmur
Malaysia

Date: July 14, 95

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Transparent BIST Scheme for Detecting *V*-coupling Faults in RAMs** submitted by **Nicole Yee-Fen Sat** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. B. F. Cockburn, Supervisor

Dr. X. Sun, Internal Examiner

Dr. J. Hoover, External Examiner

Date: July 13/95

To my parents and my sisters, for their love and support.

Abstract

A synthesizable transparent built-in self-test (BIST) scheme is developed that detects single 2, 3 and 4-cell coupling faults and two types of 5-cell scrambled pattern-sensitive faults in random-access memories (RAMs). The scheme is based on a family of deterministic tests for detecting single V-coupling faults, where V is a parameter that specifies one out of several possible fault types. The BIST design is specified in VHSIC Hardware Design Language (VHDL). A complete BIST circuit can be synthesized on the basis of two input parameters n and V, where n is the number of 1-bit storage locations in the memory. Using the Synopsys v3.1a logic synthesis tools, BIST circuits can be synthesized for n ranging from 8 bits to 64 Megabits, and for $V = 2$, 3, 4, 15 or 25. These V values correspond to single 2, 3, 4-cell coupling faults and two additional 5-cell scrambled pattern-sensitive faults, respectively. The synthesized circuit produced by the Synopsys tools is saved in Electronic Design Interchange Format (EDIF Version 2.0.0) and is imported into the CADENCE 4.2.2 computer-aided design environment. A final layout of the BIST circuit can then be generated using the standard cell AutoLayout tools in CADENCE. The behavior of the resulting BIST circuits was verified by performing Verilog simulations in CADENCE.

Acknowledgments

I would like to express my sincere thanks to my supervisor, Dr. B. F. Cockburn, for his advice, support and encouragement throughout this project. I would also like to thank the members of the examining committee, Dr. X. Sun and Dr. J. Hoover, for reviewing this thesis. Special thanks to Norman Jantz and Jeremy Sewall for technical, moral and computing support.

For financial assistance I am indebted to Telecommunications Research Laboratories and to Dr. B. F. Cockburn.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Thesis Objectives and Organization 5

2 Background **6**

2.1 RAM Architecture 6

2.1.1 Basic SRAM Cell 8

2.1.2 Basic DRAM Cell 9

2.2 Memory Failures 11

2.3 Fault Models 13

2.4 Built-in Self-test 19

2.4.1 Control Logic 19

2.4.2 Address-generation Logic 20

2.4.3 Data-generation and Response-verification Logic 20

2.4.4 Test-triggering Logic 22

2.5 Transparent BIST 22

2.6 Logic Synthesis 24

3 Test Algorithms **28**

3.1 Deterministic Tests for Detecting Single V-coupling Faults 28

3.2 $(n, V-1)$-exhaustive Codes 33

3.2.1 $(n,2)$-exhaustive Codes 33

3.2.2 $(n,3)$-exhaustive Codes 37

3.2.3 $(n,4)$-exhaustive Codes for T-neighborhoods 40

3.2.4 $(n,4)$-exhaustive Codes for Neighborhoods of Type 1 52

3.3 Transparent Near-Deterministic Tests for Detecting Single V-coupling Faults 59
3.4 An Improved Transparent Test for Detecting Single and Multiple V-coupling Faults 65
3.5 Analysis of the Probability of Aliasing 73

4 Detailed Design Description **82**
4.1 BIST RAM Architecture 82
4.2 BIST Controller 84
4.3 Data Path 93
4.3.1 Address Generator 93
4.3.2 Background Counter 94
4.3.3 Background Code Logic 99
4.3.4 Response Analyzer 105
4.3.5 Test Pattern Generator 106

5 Design Evaluation **108**
5.1 Simulation Results 108
5.2 Results of Layout Experiments 112

6 Summary and Conclusions **115**

A VHDL Code **127**

B Schematics **189**

C Simulation Waveforms **197**

List of Figures

1 Growth in DRAM Storage Capacity 2

2 RAM Architecture (one cell array) 7

3 Fully Complementary MOS Cell 8

4 Single-transistor DRAM Cell 10

5 Bathtub Curve 13

6 Example : 4-coupling Fault 15

7 5-cell Physical Neighborhood Pattern-Sensitive Fault 16

8 Logical and Physical Neighborhoods of a 5-cell NPSF Before and After Address Scrambling: (i) Before Address Scrambling; (ii) After Row Address Scrambling; (iii) After Row and Column Address Scrambling 17

9 BIST RAM Architecture [7] 19

10 Design Flow Using Logic Synthesis 26

11 Timing Diagram Notation 28

12 4×5 Background Matrix. 29

13 Test Structure Based on the 4×5 Background Matrix in Figure 12. 30

14 Deterministic Test Algorithm 32

15 $(8,2)$-exhaustive code 34

16 $(8,3)$-exhaustive code 37

17 (i) Logical Neighborhood for a Scrambled Pattern-Sensitive Fault; (ii) Corresponding Scrambled T-neighborhood 41

18 (4,3)-exhaustive Base Matrix Used in the Construction of a $(16,4)$-exhaustive Code with Respect to Scrambled T-neighborhoods . . . 42

19 Construction of a $(16,4)$-exhaustive Code With Respect to Scrambled T-neighborhoods (Step 1) 43

20 (16, 4)-exhaustive Code With Respect to Scrambled T-neighborhoods 44
21 Scrambled Physical T-Neighborhoods 46
22 Backgrounds $M_{PP} = P \times P$ and $M_{PQ} = P \times Q$ 51
23 (i) Logical Neighborhood for Scrambled Pattern-Sensitive Faults; (ii) Corresponding Scrambled Neighborhood of Type 1 52
24 Construction of a (16, 4)-exhaustive Code With Respect to Scrambled Neighborhoods of Type 1 (Step 1) 54
25 (16, 4)-exhaustive Code With Respect to Scrambled Neighborhoods of Type 1 55
26 5-cell Scrambled Physical Neighborhood 56
27 Test Structure After Applying Step 0: ALO 60
28 Test Structure After Applying Step 1, $AL1$ 61
29 Test Structure After Applying Step 2, $AL2$ 61
30 Test Structure After Applying Step 3, $AL3$ 62
31 Background Matrix for a Transparent Test 63
32 Near-deterministic Transparent Test based on the Background Matrix in Figure 31 63
33 Maximum Number of Errors Observed During a March Sequence . . 66
34 Transparent Test with Multiple Signature Generations and Comparisons 68
35 Example of a Fault that is Excited by a Background Change: (a) Fault-free Case; (b) Fault that Escapes Detection; (c) Fault-free Case with Additional Read Operations; (d) Fault Detected by Additional Read 69
36 Aliasing-free Transparent Test 69
37 Aliasing-free Transparent Test Algorithm 70

38 (i) The Exact and Approximate Escape Probability for Typical RAM Sizes for V=3; (ii) Difference Between the Exact and Approximate Escape Probability 79
39 The Escape Probability for $V = 2, 3, 4, 5t$ and $5x$ 80
40 Simplified Block Diagram of BIST RAM 83
41 The BIST Controller Flow Chart 85
42 Interface Between the BIST Controller and the Data Path 89
43 The BIST Controller State Diagram 90
44 Timing Diagram 91
45 Address Generator 93
46 Background Counter for $V = 2$ and 3 94
47 Background Counter for $V = 4$ 95
48 Background Counter for $V = 5t$ 97
49 Background Counter for $V = 5x$ 99
50 Background Code Logic for $V = 3$ 100
51 Background Code Logic for $V = 4$ 101
52 Background Code Logic for $V = 5t$ 102
53 Background Code Logic for $V = 5x$ 104
54 Response Analyzer 106
55 Test Pattern Generator 107
56 BIST Circuit Schematic 109
57 (4,2)-exhaustive Code 110
58 Area Used for BIST Overhead Approximations 113
59 Area Overhead 115
60 Test for Stuck-at Faults affecting the PASS/$\overline{\text{FAIL}}$ Output 121
61 BIST Circuit (n = 1k, $V = 3$) 190
62 BIST Controller 191

63 Address Generator 192
64 Background Counter ($V = 3$) 193
65 Background Code Logic ($V = 3$) 194
66 Response Analyzer 195
67 Test Pattern Generator 196
68 Simulation 1: Segment 1 198
69 Simulation 1: Segment 2 199
70 Simulation 1: Segment 3 200
71 Simulation 1: Segment 4 201
72 Simulation 1: Segment 5 202
73 Simulation 1: Segment 6 203
74 Simulation 1: $t = 27500$ns to $t = 32000$ns 204
75 Simulation 2: Segment 1 205
76 Simulation 2: Segment 2 206
77 Simulation 2: Segment 3 207
78 Simulation 2: Segment 4 208
79 Simulation 2: Segment 5 209
80 Simulation 2: Segment 6 210
81 Simulation 2: $t = 27500$ns to $t = 32000$ns 211

List of Tables

1 Functional Faults in DRAM 12
2 All Possible 3-coupling Faults Between Three Cells c_1, c_2 and c_3. . 36
3 Transparent Test Length 73
4 BIST Controller Inputs and Outputs 88
5 Background Number for $V = 5t$ 96
6 Background Number for $V = 5x$ 98
7 BIST Area Overhead for $V = 3$ and Typical Sizes of RAM 114
8 Area Overhead of Some Proposed BIST RAMs 119
9 Typical Test Time 120
10 Test Time for 16M RAM After Partitioning 120

List of Propositions

1 Proposition 1 34
2 Proposition 2 34
3 Proposition 3 38
4 Proposition 4 38
5 Proposition 5 44
6 Proposition 6 46
7 Proposition 7 50
8 Proposition 8 53
9 Proposition 9 55
10 Proposition 10 77

List of Theorems

1 Theorem 1 32

2 Theorem 2 69

3 Theorem 3 71

4 Theorem 4 74

1 Introduction

1.1 Overview

Random-access memories (*RAMs*) are among the most important general classes of VLSI devices. RAMs are a key high-volume component in computers and in many other digital systems. With the increase in the number of computers over the last decade, RAMs have been one of the fastest growing market segments in the semiconductor industry. During the 1980's, the world memory market grew at an average compound annual rate of 23% to reach a total volume worth U.S. $15 billion [1]. RAM circuits are important not only because of their enormous production volume, but also because they are often "technology drivers" for leading-edge VLSI technologies. Due to their simple repetitive design, memories are excellent candidates for manufacturers to explore and refine new fabrication techniques [2, 3]. Consequently, many of the new generations of semiconductor technologies are pioneered by memory designs.

The growth in the storage capacity of RAM devices has been spectacular. The storage capacity of dynamic random access memories (*DRAMs*) has quadrupled every 3 years and this trend has continued through to the current 16 Megabit DRAM generation. If this trend continues, the manufacturing of 1 Gb DRAMs with 0.15 μm design rules will start by the year 2000, as shown in Figure 1 [4]. Hitachi and NEC have already announced prototype 1 Gb RAMs at the 1995 IEEE International Solid-State Circuits Conference [5, 6].

In order to ensure the correct functioning of RAMs, testing is required to verify that each manufactured RAM is functioning within its specification. There are two conflicting requirements in testing semiconductor memories. The first requirement is to achieve the lowest possible test cost, which usually means minimizing the test

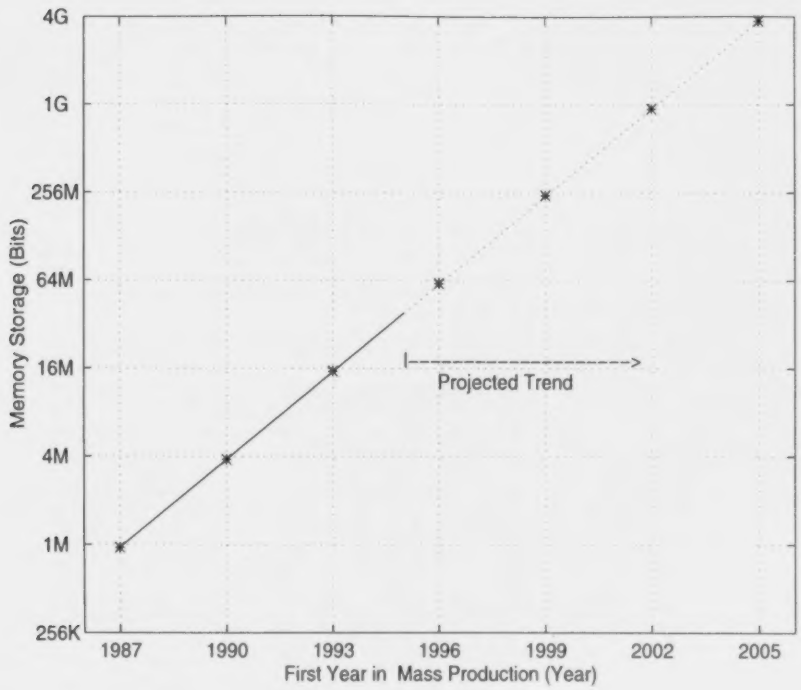


Figure 1: Growth in DRAM Storage Capacity

time. The second requirement is to achieve high fault coverage. In order to reduce cost, test length cannot be allowed to increase significantly. On the other hand, high fault coverage — which determines the ultimate quality of shipped RAMs — calls for sufficiently long tests. Therefore, we have to balance both requirements when designing an efficient RAM test.

In conventional testing environments with external testers, test vectors are generated and/or stored in automatic test equipment (*ATE*). To test a RAM, these vectors are applied via the chip's input/output pins. The main disadvantage in external testing is that as memory sizes increase, applying the vectors becomes extremely time consuming since only one location in the RAM can be accessed at a time. Furthermore, in embedded RAMs (i.e. RAMs surrounded by other logic

circuitry), one can often not access the address, read/write and data lines directly. In such cases, the test data may have to be sent through serial scan chains, which increases test time significantly [7]. Alternately, the test data may have to be propagated in several clock steps through the intervening logic.

To ease the task of designing a test algorithm, memory failures are modeled as functional models that describe how the failures behave in the boolean domain. These models are convenient because they are relatively independent of the memory design and technology. Ideally, a fault model should be based on the results of experimental studies of the actual physical defects and faulty behaviors that occur in real RAMs. Some of the models that are widely used in memory testing include stuck-at faults, coupling faults and pattern-sensitive faults [8]. These faults will be reviewed in detail in Chapter 2.

A memory test algorithm can be deterministic or probabilistic. *Deterministic* tests guarantee 100% fault coverage over an assumed fault model. This means that all the faults that behave in the way described by the fault model will be detected. A *probabilistic* or random test which uses pseudorandom techniques is expected to give high, but not necessarily 100% fault coverage. The advantage of probabilistic tests is that they can be entirely independent of the fault model.

A memory is tested by observing its response to a set of test vectors. The test vectors are applied to the memory using write operations. The output response is obtained by reading the contents of memory cells and comparing them with the expected values. In probabilistic testing, the output response of the RAM under test is compressed into a binary signature which is compared to an expected signature of a good memory (which may be the signature found using a known good part, or the signature from a simulation model). This method of test response compaction is called *signature analysis* [9]. The signature is obtained by feeding the response of the RAM into a linear feedback shift register (*LFSR*). This method

is very efficient due to the low hardware involved. The only limitation of signature analysis is aliasing, which can occur due to the information loss inherent in any data compaction technique. Aliasing is the situation when the signature obtained from a faulty memory is the same as the fault-free signature. (When aliasing occurs, the effects of the errors have cancelled out in the signature.) However, recent work has shown that aliasing-free signature analysis can be achieved in RAM BIST [10] if the maximum number of errors that can be created by a fault is known ahead of time. By examining the error patterns generated by a given class of faults, an aliasing-free signature analyzer (or LFSR) can be designed.

Built-In Self-Test (*BIST*) refers to the addition of on-chip test circuitry to facilitate testing [8].[1] A BIST scheme for RAM is a scheme where the test mechanism for testing the memory is completely contained within the chip itself. In a BIST environment, test data can be generated on-chip thus solving the test data storage problem. Test time is also reduced since we no longer need to transmit the test vectors via the relatively slow I/O pins. In the case of embedded RAMs, BIST enables us to bypass scan chains or intervening logic and access the RAMs directly.

Transparent BIST was first introduced in a restricted form by Koeneman in 1986 [11]. If a RAM is fault-free, the contents of the RAM after applying a *transparent* BIST algorithm will be the same as its initial contents. This feature makes transparent BIST more attractive than standard BIST since it can be used for both fabrication testing and periodic testing of a memory that is holding live data in a running system. Nicolaidis proposed a technique which allows most test algorithms to be transformed into transparent BIST algorithms [12]. We use a modification

[1] Another definition of BIST is to enhance the functionality of a logic circuit to test itself [9]. One may argue that the term Built-In Testing (*BIT*) is more appropriate in memory testing since the test circuitry is used to test the memory and not itself. We will use the term BIST as defined in [8] (i.e. the BIST circuitry does not have to test itself).

of Nicolaidis' method in our BIST RAM scheme.

1.2 Thesis Objectives and Organization

This thesis involves the design and characterization of a synthesizable built-in self-test scheme for detecting single 2, 3, and 4-cell coupling faults and two additional 5-cell scrambled pattern-sensitive faults in RAM.

The objective of this project is to demonstrate :

- a tool for automatically generating the logic and the layout of BIST circuits to test for the above faults.
- a scheme that is aliasing-free for the case of single faults. In addition, the probability of aliasing for the case of multiple faults is very low.
- the overhead cost of the BIST scheme in terms of area is low.

This thesis is divided into six chapters. Chapter 1 gave a brief overview of memory testing and the different kinds of RAM test. In Chapter 2, further background material is presented. Chapter 3 gives a detailed description of the structure of the test algorithms that are implemented in our new BIST RAM scheme. An analysis of the probability of aliasing is also given in this chapter. Chapter 4 discusses in detail the design of the BIST RAM. In Chapter 5, we present the results of simulations and layout experiments. Finally, we summarize and conclude the thesis in Chapter 6.

2 Background

There are six sections in this chapter. In Section 2.1 the architecture and operation of RAMs are discussed. In Section 2.2 we survey some of the failures that occur in memories. Section 2.3 presents some of the models used to describe memory failures. In Sections 2.4 and 2.5 we review BIST architectures and introduce the concept of transparent BIST. Section 2.6 explains the logic synthesis process and highlights some of the advantages of synthesis over conventional gate-level design methods.

2.1 RAM Architecture

The key architectural elements of a RAM consist of one or more rectangular cell arrays, decoder logic, sense-amplifiers and read/write logic. Figure 2 shows a simplified general model of a RAM.

The cell array contains $2^N \times 2^M$ individual storage cells. Each cell stores one bit of information. The address, which is $N + M$ bits long, is used to uniquely identify each cell for reading and writing. The higher order N bits of the address are connected to the row decoder, which selects one row of cells out of 2^N possible rows by asserting the corresponding word line signal. The lower order M bits go to the column decoder, which selects the desired columns out of 2^M possible columns by creating two connections to the corresponding pair(s) of bit lines for the selected column(s) of cells. The number of columns that are selected at a time depends on the number of data lines in the RAM chip, that is, the number of cells that can be accessed together during a single read or write operation. It is common to have several cell arrays on a RAM chip. In this case, the address has additional bits for identifying the one cell array which contains the cell to be accessed. The advantage of having multiple cell arrays is that the effective length of the word and

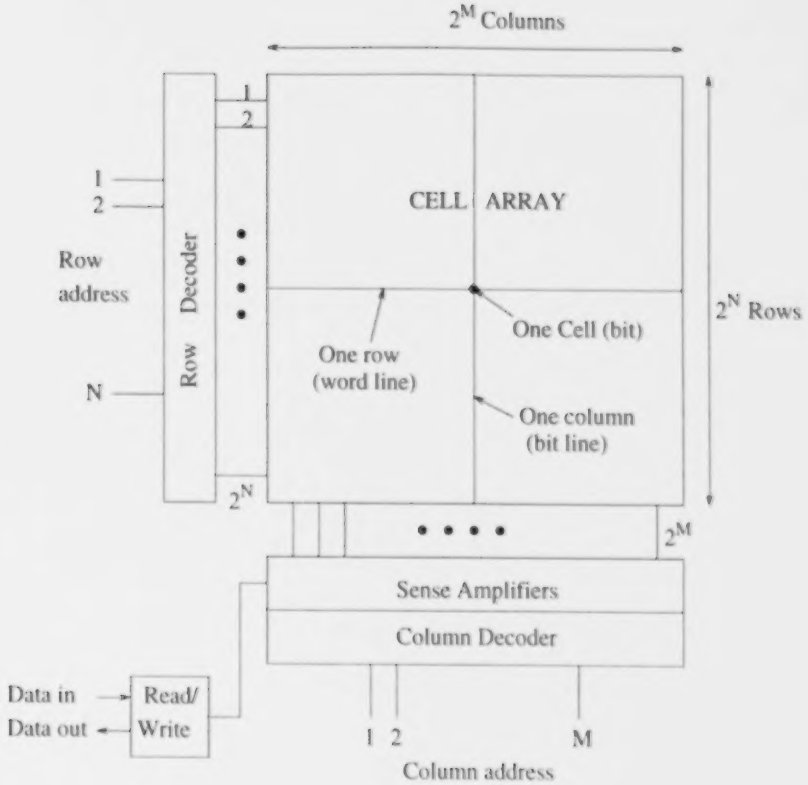


Figure 2: RAM Architecture (one cell array)

bit lines is reduced. This will reduce the access time of the RAM by reducing the amount of line capacitance that must be charged/discharged during the memory operation.

Two types of RAM, static RAM (*SRAM*) and dynamic RAM (*DRAM*), will be discussed in the following two subsections. The basic difference between the two types of RAM is that in SRAMs, data is stored in latches while in DRAMs, data is stored as charges on capacitors. Also, read operations in a SRAM are non-destructive, and data is retained as long as power is applied to the memory. In the case of a DRAM, the read operation discharges the storage capacitor and is

therefore destructive with respect to any data that was stored. DRAM cells also have to be refreshed periodically since the charge stored on the capacitors tends to leak away with time.

2.1.1 Basic SRAM Cell

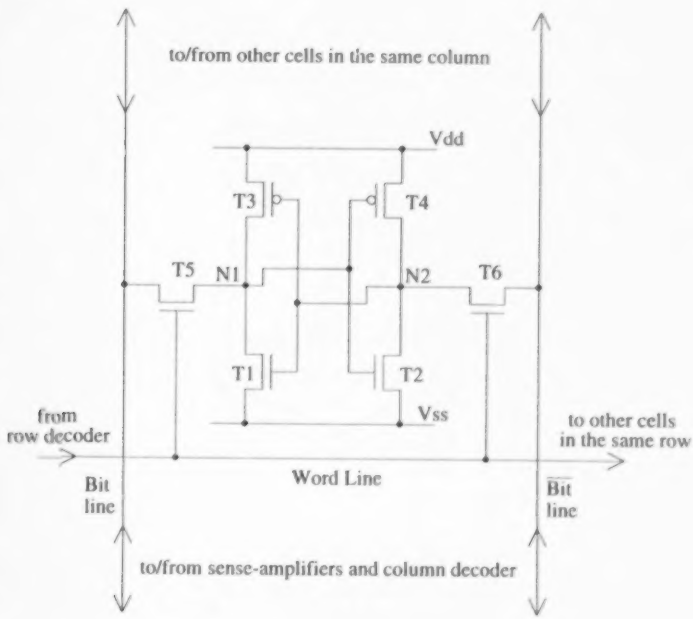


Figure 3: Fully Complementary MOS Cell

Figure 3 shows a six transistor, fully complementary static CMOS memory cell. Information is stored in the form of voltage levels in the latch which is formed by two cross-coupled inverters. The latch has two stable states, 1 and 0. If the cell is in state 1, N1 is at logic level 1 and N2 is at logic level 0. In this case, T1 and T4 are off while T2 and T3 are on. If the cell is in state 0, N1 is at logic level 0 and N2 is at logic 1 so that T1 and T4 are on while T2 and T3 are off. The state of an SRAM cell will stay the same as long as power is maintained across Vdd and Vss,

and the state of the cell is not changed by write operations addressed to the cell.

The memory cell array is composed of an array of SRAM cells which are accessed using a row decoder and a column decoder. Each column of cells transmits differential data signals along a shared vertical pair of bit lines. To read an SRAM cell, the *bit* and $\overline{bit}$ lines are precharged to high. (Precharging is a technique that speeds up access time.) The row decoder asserts the word line high so that the NMOS pass transistors T5 and T6 are on. At this time, one of the two cross-coupled inverters in the enabled cell in the column will pull one of the bit lines low. The different voltage levels in *bit* and $\overline{bit}$ is detected and amplified by a sense-amplifier which is connected across the bit lines at the edge of the cell array. Since the column decoder is selecting this particular sense-amplifier, the recovered data signal is driven to the data output port of the RAM.

To write a logic value into a SRAM cell, *data* is driven by the powerful sense-amplifier buffers onto the *bit* line while $\overline{data}$ is driven simultaneously onto the $\overline{bit}$ line. The one word line corresponding to the addressed cell is activated by the row decoder so that T5 and T6 are turned on. The bit line driver then forces N1 to *data* while the $\overline{bit}$ line driver forces N2 to $\overline{data}$ so that the new data is stored in the latch.

2.1.2 Basic DRAM Cell

Figure 4 shows a single-device DRAM cell which consists of an enhancement mode select transistor Q and a storage capacitor C. The gate of the select transistor is controlled by a word line. In DRAMs, the cells in one column share one bit line. The plate node is common to all cells in the array and provides a reference potential. A read operation proceeds as follows: the bit line is precharged to a reference level, which is usually mid-way between logic high and low. The word

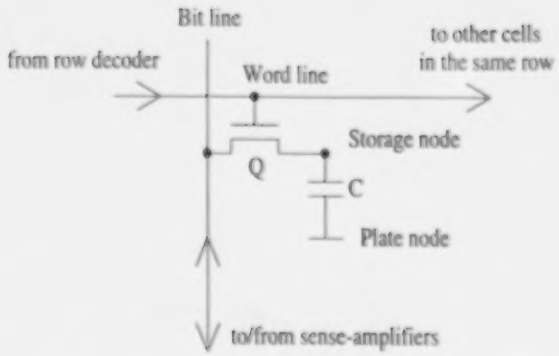


Figure 4: Single-transistor DRAM Cell

line is activated by the row decoder which turns the access transistor Q on. This causes the charge stored on the capacitor to be dumped out onto the bit line. A sense-amplifier located at one side of the cell array compares the voltage level on the bit line with a reference voltage to determine if the DRAM cell stored a logic 1 or a 0. This read operation is destructive since data is no longer stored in the cell after C discharges. The earlier state of the cell is then restored in a write-back operation. The sense-amplifier accomplishes this operation by amplifying the voltage level on the bit line so that C is charged back to its initial level via the bit line and the still-enabled access transistor Q. The read operation ends when the word line signal is de-asserted.

A write operation is very similar to the write-back operation except the bit lines are forced to the level corresponding to the externally supplied data rather than the data originally stored in the cell. The word line is then activated by the row decoder so that C is charged to the voltage level on the bit line. The write operation ends when the word line signal is de-asserted.

2.2 Memory Failures

VLSI fabrication is a complex, multi-step process which involves growing pure silicon crystals, depositing photolithographic or photoresist material, aligning masks, etching, implanting ions and oxidizing. A physical defect which can affect the functionality of an IC's can be introduced in any of these processing steps. Physical defects are typically caused by the presence of dust particle on the silicon wafer or the masks, mask misalignment or mask imperfections [8].

In RAMs, physical defects can occur in peripheral circuits (address decoder, read/write logic) and in the memory cell array. Defects in the peripheral circuits cause faults less frequently because the density of the minimum-width structures is much smaller than the density in the cell array [13]. Also, the effects of such defects are usually catastrophic and easily detected by any reasonable test. Consequently, most of the research work in memory testing is focused on defects that occur in the memory cell array. Table 1 shows a list of physical defects and failures that can occur in RAMs [8].

A memory failure occurs when a physical defect in the RAM causes it to behave in a manner that violates its specification. Theory and experience have shown that the failure rate for semiconductor devices normally follows the well-known bathtub curve [3] (see Figure 5). There are three different periods :

- *infant mortality*: a period with an initial high failure rate normally due to manufacturing defects, such as gate-oxide shorts.
- *Normal life*: a stable period with low failure rate (due to random failures) representing the useful time period of device life.
- *Wear out*: after the normal life period, the failure rate increases rapidly due to various aging mechanism, such as electromigration and corrosion.

Cell stuck
Write driver stuck
Read/write line stuck
Data line stuck
Data line open
Short between data lines
Crosstalk between data lines
Address lines stuck
Address lines open
Open decoder (no access possible)
Wrong access due to decoder fault
Multiple access due to decoder fault
Cell can be set to 0 but not to 1(or vice versa)
Pattern-sensitive interaction between cells

Table 1: Functional Faults in DRAM

Burn-in testing is used by manufacturers of RAMs to increase the reliability of memory chips. During the burn-in period, RAM chips are subjected to high temperature to accelerate the development of failures due to latent manufacturing defects, i.e. to accelerate passage through the infant mortality region. At the end of the burn-in period, the memories are tested to determine if they are faulty. This way, the manufacturers can ensure that their customers only receive memory devices that are at the beginning of their normal life.

Figure 5: Bathtub Curve

2.3 Fault Models

Physical failures that occur in RAMs are often mathematically modeled at the transistor, gate or functional levels of abstraction. A *fault model* can describe a set of different physical faults that have the same faulty behavior. Many fault models have been developed at the functional level. Functional fault models are convenient because they can represent faults in many different technologies.

There are two assumptions that are commonly used in the development of RAM fault models: the single fault assumption and the nondestructive (or fault-free) read operations assumption. The *single-fault assumption* reduces the complexity of the test procedures by assuming that at most one fault can be present in a RAM. This assumption is justified since most tests that detect all single faults in a fault model often also detects most multiple faults [7]. The *fault-free read assumption*, where read operations are assumed not to trigger errors, is also used for practical reasons. Most test procedures apply a sequence of write operations and use the read operations to verify the success of the write operations. Tests would be much

harder to design and analyze if read operations could cause errors. In the case of SRAMs, the electrical effects of read operations should be weaker than those of write operations, so the assumption seems reasonable. The assumption is not as appropriate for DRAMs, where each read is followed by a write.

The following classes of faults are widely used in memory testing:

- *Stuck-at faults*: a memory cell is said to be stuck-at-1 (stuck-at-0) if its contents remain fixed at logic 1(0), irrespective of what is written to the cell.

- *Coupling Faults*: a pair of memory cells i and k is said to be coupled if a write which changes the contents of cell i also causes the contents of the second cell k to be forced to 0 or 1. There are two types of coupling faults: an *idempotent coupling fault* is one in which a state transition in cell i causes cell k to be forced to a certain value (0 or 1). An *inversion coupling fault* is one which the appropriate transition in cell i causes the contents of cell k to invert. The effects of two coupling faults can cancel out; for example, if cell k contains a logic 1 and is coupled to both cell i and a third cell j, then the effect of a transition in cell i, which causes cell k to go to 0 can be cancelled by a transition in cell j which causes cell k to go back to 1. The possibility of error cancellation increases the difficulty of detecting multiple faults.

  Single coupling faults can also be defined which involve three or more cells. The fault model for coupling faults involving $V \geq 2$ memory cells is the $V-$*coupling* fault model [14]. A V-coupling fault behaves as follows: A transition in one cell C_1 changes the content of a second cell C_2, provided the remaining $V-2$ cells $C_3, C_4 \ldots C_V$ store a specific pattern of 1's and 0's. Figure 6 shows an example of a 4-coupling fault involving four cells in memory. The content of the four cells are a, b, c and d, where $a, b, c, d \in \{0, 1\}$. In the case of a good memory, when a write of value $\bar{c}$ to cell i is

applied, only the content of cell i is changed from c to $\bar{c}$. However, in the case of a bad memory, when the same write operation is performed, the content of a second cell k is changed from d to $\bar{d}$. However, this faulty behavior only occurs if the remaining two cells store values a and b.

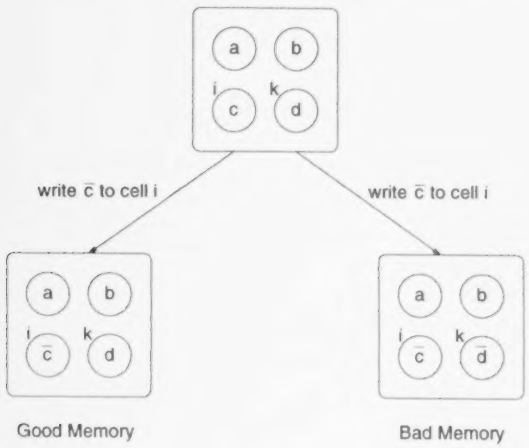


Figure 6: Example : 4-coupling Fault

- Pattern-Sensitive Faults (*PSF*): a memory cell is said to have a pattern-sensitive fault if its state is altered, or changes in its state, are inhibited by either :

  1. a pattern of 0's and 1's in surrounding cells,
  2. 0 to 1 and/or 1 to 0 transitions in surrounding cells.

As RAM density increases, the cells become physically closer, and PSFs are expected to become the predominant faults [15]. However, testing for unrestricted PSFs, in which the patterns can involve any n cells in the RAM, requires an $O(2^n)$ test and is therefore impractical given the sizes of n for modern memories [8, 16]. Most test algorithms for PSFs consider a subset of PSFs called Neighborhood

Pattern Sensitive Faults (*NPSFs*). A *neighborhood* is the set of cells involved in a particular fault. Figure 7 depicts a 5-cell neighborhood within a 6×6 cell array. The *base cell*, labeled B, is the cell whose contents are disturbed by the fault. The four physical neighbors of the base cell are often called N (North), E (East), W (West), and S (South).

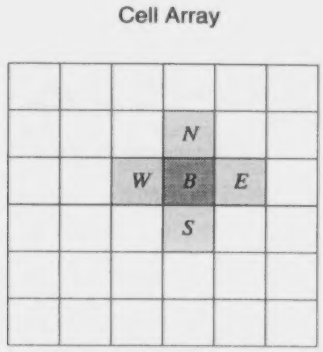


Figure 7: 5-cell Physical Neighborhood Pattern-Sensitive Fault

Tests for detecting NPSFs are difficult to construct in practice because the mapping from cell addresses to cell locations in the memory array is not often known. This is because RAM chips are usually designed with spare rows and columns to improve the yield. This redundancy allows a memory to be repaired (if necessary) by the manufacturer. Memories are repaired by disconnecting the rows and/or columns that contain defective cells, and replacing them with spare rows and/or columns. After a memory chip is reconfigured, physically adjacent cells may no longer have consecutive addresses [17]. In addition to redundancy, a test designer cannot assume that the logical and physical addresses of a memory are identical because memory designers sometimes scramble the order of the address lines for reasons of layout convenience [17]. Figure 8 shows how address scrambling changes the physical neighborhood for a 5-cell logical neighborhood. Before address

scrambling, the logical neighbors of base cell B, N, E, W and S, are the same as its physical neighbors, as shown in Figure 8(i). After the row addresses are scrambled, (Figure 8(ii)), the N and S cells are no longer the physical neighbors of B. Finally, after both the row and the column addresses are scrambled (Figure 8(iii)), none of the logical neighbors of B are still physical neighbors.

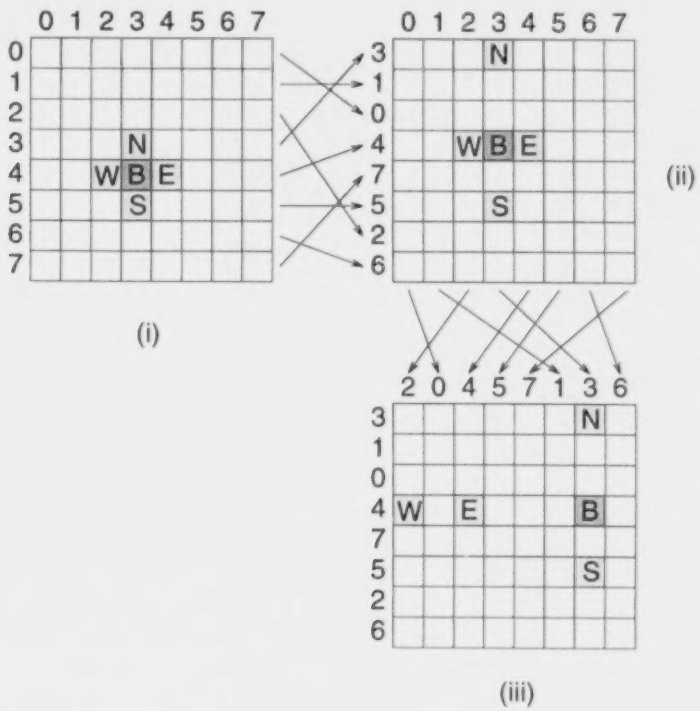


Figure 8: Logical and Physical Neighborhoods of a 5-cell NPSF Before and After Address Scrambling: (i) Before Address Scrambling; (ii) After Row Address Scrambling; (iii) After Row and Column Address Scrambling

Franklin and Saluja [18] observed that, even after scrambling both the row and column addresses, the physical N and S neighbors of the base cell have the same logical column address as the base cell, even though they are no longer the logical N and S neighbors. Similarly, the physical W and E neighbors have the same

logical row address as the base cell [18]. This means that, in the absence of the exact scrambling mapping, a test for detecting the scrambled 5-cell NPSF has to consider all possible 5-cell logical neighborhoods involving a base cell B, two other cells in the same row as B, and two other cells in the same column as B.

Before address scrambling, the number of 5-cell neighborhoods which have to be considered is equal to $(\sqrt{n}-2)^2 \approx n$. Here we assume that the memory cells are arranged in a $\sqrt{n} \times \sqrt{n}$ square grid and that the cells at the edges of the grid are not considered as potential base cells. After address scrambling, all n cells in the memory can be base cells. For a base cell B, the N and S neighbors can be any two cells from the remaining $\sqrt{n}-1$ cells in the same column. Similarly, the E and W neighbors can be any two cells from the remaining $\sqrt{n}-1$ cells in the same row as B. Therefore, there are $\begin{pmatrix} \sqrt{n}-1 \\ 2 \end{pmatrix} \times \begin{pmatrix} \sqrt{n}-1 \\ 2 \end{pmatrix}$ possible combinations of N, E, W and S neighbors for a single base cell. Since all n cells in the array can be base cells, the number of 5-cell neighborhoods which have to be considered is equal to $\frac{n}{4}(\sqrt{n}-1)^2(\sqrt{n}-2)^2 \approx n^3$. Fortunately, the test times for detecting scrambled PSFs do not grow with n^3 if numerous faults can be tested in parallel.

Most fault models assume that the effects of the defects in the address decoder and read/write logic map to faults in the cell array. In other words, the general assumption is that, when tests of the memory cell array are applied, the faults in the decoder and read/write logic will behave as faults in the memory array [14, 8]. For example, a stuck-at fault in the read/write logic will appear as a group of memory cells with stuck-at faults. Similar arguments can be used for the decoder faults as well. Therefore, an algorithm that detects all memory cell stuck-at faults will also detect most decoder and read/write logic faults. We will call this third widely used assumption the *fault mapping assumption.*

2.4 Built-in Self-test

The basic BIST RAM architecture requires the addition of four hardware blocks to the core RAM: control logic, address-generation logic, data-generation and response-verification logic, and test-trigger logic [7]. The block diagram of a general BIST RAM architecture is shown in Figure 9.

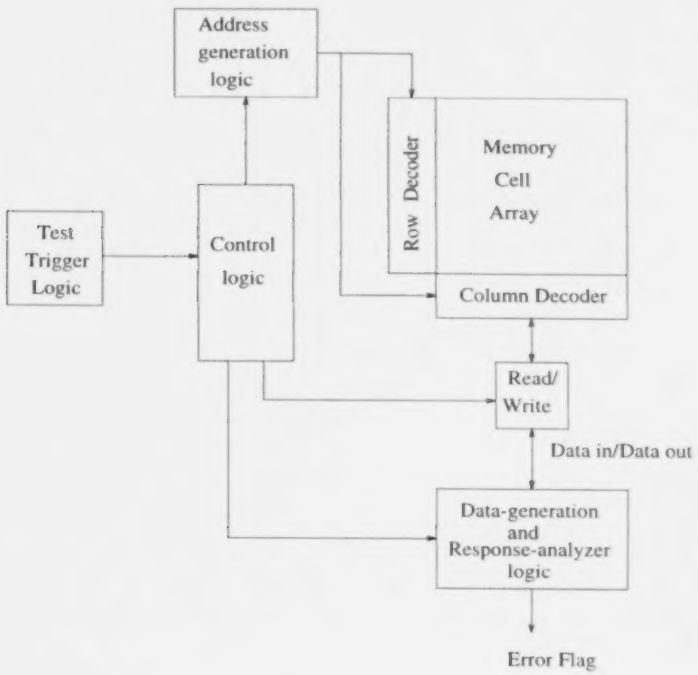


Figure 9: BIST RAM Architecture [7]

2.4.1 Control Logic

The control logic activates a test in response to an external command and controls the flow of the test algorithm. It can be implemented using random logic or microcode. Random logic is faster and often has less area overhead than microcode-based design [7]. However microcode is becoming more popular since it provides

flexibility as well as ease of implementation due to its regular structure. For large RAMs, microcode-based control has been shown to have an area overhead which does not exceed that of random-based logic design [19]. On the other hand, logic synthesis makes microcode design less attractive than random logic because the latter can be easily generated using synthesis tools. Furthermore, logic synthesis enables random-based designs to be parameterized which essentially provides the same flexibility as in microcode. For example, a single parameter *Test* can be used to selectively generate the circuits for the random-based control logic, where different values of *Test* correspond to different test algorithms that are used in the BIST RAM. Logic synthesis will be described further in section 2.6.

2.4.2 Address-generation Logic

Address-generation logic is used to generate the address sequence required by the test algorithm. This is usually accomplished using counters or linear-feedback shift registers (*LFSRs*). The control logic provides control signals (e.g. increment or decrement signals for a counter) to the address-generation logic to obtain the address sequence required by the test algorithm. LFSRs are often more area-efficient than counters [12]. Thus, when the address sequence is unimportant, LFSRs are often preferred.

2.4.3 Data-generation and Response-verification Logic

The data-generation logic produces test patterns which are applied to the memory cell array. These patterns vary according to the test algorithms.

The response of the RAM under test (the sequence of read values) can be verified by comparing against the expected response. For a given set of test patterns, a set of expected responses can be obtained by simulating a fault-free instance of the

RAM under test. These responses can be stored in an on-chip read-only memory (*ROM*), but this scheme would require too much area overhead to be practical, especially when testing large RAMs. The expected response can sometimes be generated at test time using data-generation logic. This method alone cannot be used in transparent testing, however.

An alternative method, which does not require storing the complete response of a good RAM, is to use a data compaction technique called *signature analysis*. In this method, the read values are fed into a signature analyzer which produces a relatively short binary sequence, called a *signature*, by the end of the test. The signature analyzer can be an LFSR or a multiple-input signature register (*MISR*) [9]. (Feeding a bit stream into an LFSR is mathematically equivalent to polynomial division in the binary field. The content of the LFSR after this operation is the remainder of the polynomial division [9].) The compacted signature is compared to a known fault-free signature, usually obtained by performing simulations on a fault-free RAM. The result of this comparison is used to indicate if the RAM under test is faulty. The advantage of using signature analysis is that the LFSR or the MISR required for this kind of compaction is easy to implement. In addition, the area overhead introduced in this scheme is minimal compared to using a ROM. However, signature analysis introduces an *aliasing* problem. Aliasing occurs when a faulty RAM produces a response sequence which is different from that of a fault-free RAM, but the resulting compacted sequences are identical. Aliasing is inevitable whenever a data compaction technique is employed. For a k-bit MISR, the probability of aliasing usually approaches 2^{-k} [9] (assuming that $l >> k$ where l is the length of the information bits being compacted). Therefore, the problem of aliasing can be reduced to acceptable levels by using a sufficient long MISR, say $k = 16$.

2.4.4 Test-triggering Logic

All BIST RAMs have two modes of operation: a normal mode and a test mode. In the normal mode, the BIST circuitry is de-activated and the RAM performs normal read and write operations. In the test mode, the BIST circuitry is active and the test algorithms are performed on the RAMs. The test mode is entered by using overvoltages, extra package pins, or a special timing sequence [7]. Using unique timing sequences is better than overvoltages and extra package pins because overvoltages require the generation of an additional voltage signal, while extra package pins increase the size and cost of the RAM chip. This is of course not an issue in embedded RAMs.

2.5 Transparent BIST

RAMs are one of the main components in many digital systems. In some applications, RAMs are tested periodically to increase the reliability of the running systems. In such cases, the test algorithm must not destroy the contents of the RAMs. One way of satisfying this requirement is by saving the contents of a RAM before a test algorithm is applied. The contents can be stored in a spare chip, and later written back to the RAM after the test is completed. However, if the RAM is embedded in a VLSI circuit, it is often difficult to access the address, read/write and data lines from outside the circuit. Also, saving and restoring the contents of an embedded RAM can become very difficult and may require another spare RAM to be implemented in the circuit. For large RAMs this is often impractical. Transparent BIST solves this problem efficiently since the contents of a RAM after a transparent BIST algorithm is applied will be the same as its initial contents if the RAM is fault-free.

Koeneman introduced the concept of transparent BIST in 1986 [11]. He derived

a technique based on the linearity of signature analyzers. As noted by Nicolaidis in [12], Koeneman's original method has four main drawbacks:

- This technique works only if the output response verification is performed by linear compaction.
- Aliasing may occur where an error can be masked and thus escape detection.
- Test patterns must be composed of elementary loops which perform exactly one read and one write operation in each cell. Most existing algorithms do not have this particularly simple structure.
- The technique can reduce the fault coverage since it requires modification of the test algorithm.

Nicolaidis derived a series of steps which can be used to transform most test algorithm into a transparent BIST algorithm [12]. These steps modify an algorithm so that when the algorithm is applied to a RAM, the data in each RAM cell is complemented an even number of times. This ensures that if a cell is fault-free, its value at the end of the test will be the same as its initial value.

In Section 2.4.3, we described the task of the response-verification logic in a BIST design. Essentially, the correctness of the read values can be verified by signature analysis, where the expected signature is determined by simulation. This approach does not work with transparent BIST because the correct signature depends on the unpredictable contents of the RAM before the test is applied. Nicolaidis solves this problem by including a signature generating step as the first phase of the transparent BIST test. In this phase, only the read operations are applied to the RAM to compute the expected signature. In the second phase of the test, both the read and write operations are performed and a second signature is

produced. Extra logic is needed to selectively invert some of the values read from the RAM under test in the first phase so that the input to the signature analyzer is the same as the input obtained when the second phase is applied to a fault-free RAM. The second signature is then compared with the first signature, and the result of this comparison indicates if the RAM is faulty. For most fault models, Nicolaidis' transformation does not cause a reduction in fault coverage beyond the loss caused by the aliasing in the signature analyzer. However, aliasing-free signature analyzers can be designed, using a further technique described in [10] if the maximum number of errors in the response is bounded.

2.6 Logic Synthesis

The density (the number of devices) which can be integrated onto a single chip has been increasing rapidly as VLSI technology advances. Hardware designers are faced with the task of designing larger and more complex logic circuits, a task that is complicated, time consuming, and error-prone. This has resulted in a need for design automation [20]. The goal of design automation for electronic systems is to fully automate the transformation of a specification given at a high level of abstraction e.g. a systems behavioral information, into a low level description, e.g. a mask geometry which can be fabricated. *Synthesis*, which is the process of translating a behavioral description of an electronic circuit into a lower level description, usually a gate level description, of the circuit [21] plays a major role in design automation. There are three main components involved in synthesis: hardware description languages, the circuit design itself, and the technology used for the implementation of that design. Hardware description languages *(HDLs)* are used to specify the desired behavior of a circuit. Some of the more common HDLs include VHDL (VHSIC hardware description language) and Verilog HDL.

The target technology provides information on the standard cells for programmable modules that can be used in the design and fabrication processes. Figure 10 shows a typical synthesis design flow. At the top level, a designer translates an idea into a behavioral specification using HDL. This specification is the input to a behavioral synthesis tool which produces an unoptimized logic description of the initial design. Next, the logic description is optimized using automated heuristic-guided circuit simplification rules and mapped to a given library of primitive logic parts in a particular target technology. Using layout tools, the optimized logic description or netlist is further translated into a layout description which can be fabricated.

The main advantage of synthesis is that it enables a circuit designer to work at a higher level of abstraction. Detailed descriptions of the internal logic gates and interconnections are not required. Instead a designer only has to provide the functionality and the constraints of a circuit. This undoubtedly leads to shorter circuit descriptions and therefore eases the circuit debugging and updating processes. The result is a faster turnaround time from initial design idea to circuit implementation.

A second advantage of synthesis is that it allows a designer to evaluate different design alternatives stemming from the same initial specification. These designs can be generated quickly, and their performance can be determined and compared.

Having tools to generate circuits automatically can often lead to more compact and efficient circuits than the circuits obtained by implementing the design in the traditional ways using CAD tools. For example, the initial BIST circuits I designed at the conventional gate-level using CADENCE were about 10% larger in terms of area overhead than the circuits I generated later using the synthesis tool in Synopsys. As systems become increasingly complex, it will become more and more difficult to create their design by hand since the number of objects to be handled will too high. Synthesis tools with algorithms for optimizing the gate

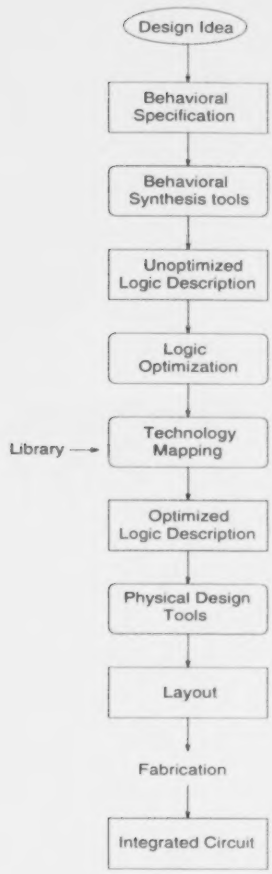


Figure 10: Design Flow Using Logic Synthesis

level description of a circuit can usually minimize the number of gates required for a particular design [21].

The ability to parameterize a circuit design is another advantage of synthesis. For example, a VHDL specification for a 4-bit counter is very similar to a specification for a 5-bit counter; the only difference is that in the former, the counter increments to a maximum of 2^4, while the latter counts up to 2^5. By parameterizing the counter circuit, only one VHDL description of the counter is required,

i.e. a behavior description of a counter that counts up to N, where N is the input parameter of the counter circuit. The circuit designer can then synthesize different counter circuits using the same VHDL description for a generic counter.

Using synthesis, a tool for generating BIST circuits automatically can be developed. The behavior of each of the hardware blocks in a BIST circuit: control logic, address-generation logic, data-generation and response-verification logic, and test-trigger logic can be described using VHDL. The different blocks can then be linked together to form the overall BIST circuit. With parameterization, each block can be described in such a way that the width of the data path is not defined. This will enable the BIST circuit generator to produce BIST circuits for different sizes of RAMs and for detecting different classes of faults. A memory designer can use such a tool to increase the reliability of his/her designs with minimal knowledge of the details of the BIST circuits.

3 Test Algorithms

There are five main sections in this chapter. Section 3.1 describes the structure of the deterministic test for detecting single V-coupling faults. In section 3.2, the properties and the proofs of the $(n, V-1)$-exhaustive code constructions used in the deterministic tests are discussed. In section 3.3, we transform the deterministic tests into transparent near-deterministic tests using Nicolaidis' technique. A further improved transparent test, which is aliasing-free for single faults, is described in section 3.4. Section 3.5 analyzes the probability of aliasing for multiple faults.

3.1 Deterministic Tests for Detecting Single V-coupling Faults

Cockburn proposed a family of deterministic tests for detecting single V-coupling faults for $V \geq 2$ [22]. The structure of the tests is determined by an $n \times m$ binary matrix, also called a *background matrix*, where n is the number of words in the RAM under test and m is the number of backgrounds used in the test. Timing diagrams will be used to describe the structure of these deterministic tests.

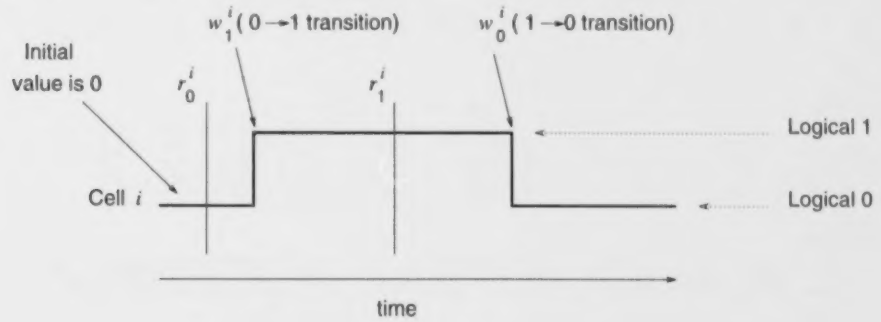


Figure 11: Timing Diagram Notation

Consider the timing diagram in Figure 11. The bold-line waveform shows the

content of a memory cell i over time. A read operation is denoted by a vertical line. When the content of the cell is zero, the vertical line indicates a read with the expected value 0; this is denoted symbolically by r_0^i. When the content of cell i is 1, the vertical line denotes a read with the expected value 1, denoted r_1^i. A write operation which changes the content of the cell from 0 to 1, denoted by w_1^i, is shown as a $0 \rightarrow 1$ transition. Finally, a write that changes cell i from 1 to 0, w_0^i, is shown as a $1 \rightarrow 0$ transition.

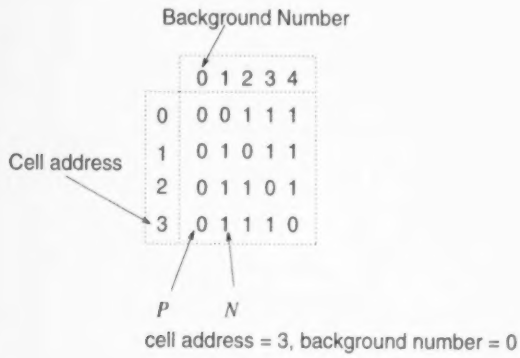


Figure 12: 4×5 Background Matrix.

Figure 12 shows a background matrix with $n = 4$ and $m = 5$. Each column in the matrix is a background that is loaded into the RAM. The *background number* is used to indicate the current background (matrix column). The *row pointer* is the cell address. For every background, each memory cell has a present bit value, P, and a next bit value, N. In the example shown in Figure 12, the cell address is equal to 3 and the background number is equal to 0. The corresponding P bit for cell 3 is the bit in the fourth row and the first column, which is equal to 0. The N bit is the bit in the same row as P but in the next column, i.e. background number + 1. This column is called the next background. In this example N is equal to 1. Similarly, if the background number is equal to 3, and the cell address

is equal to 2, the P bit is 0, and the N bit is 1.

The corresponding timing diagram for the test generated from the 4×5 background matrix in Figure 12 is shown in Figure 13. In this example, the RAM under test has four cells since n is equal to 4. There are five regular sequences of read and write operations which are called *marches*. In Figure 13, the marches are labeled March 1, March 2, ..., March 5. Each march corresponds to operations that are applied with respect to one background in the background matrix, i.e. March 1 corresponds to background number = 0, March 2 corresponds to background number = 1, and so on.

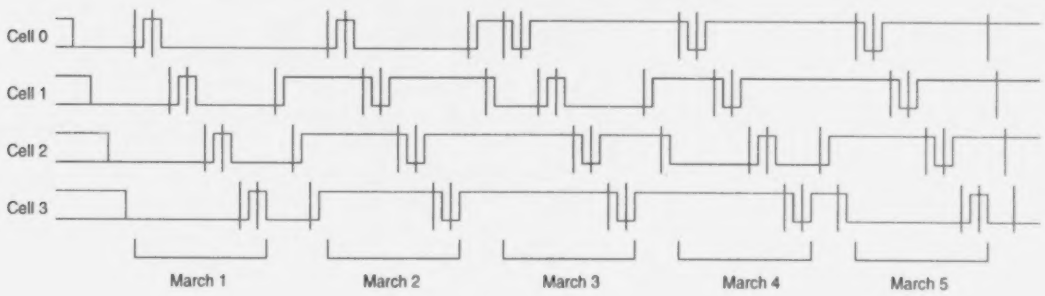


Figure 13: Test Structure Based on the 4×5 Background Matrix in Figure 12.

At the beginning of the test, the background number is set to 0, indicating that the current background is now equal to the first column in the background matrix. Note that the cells are assumed to contain either 0 or 1 at the beginning of the test. The RAM under test is then initialized (i.e. written) according to this background. In this example, since the first column is the all-zero column, all the cells in the 4×1 RAM are written to zero, in ascending address order. The ascending address order is denoted symbolically by the $\Uparrow$ symbol. Therefore the initialization step can be written as $\Uparrow (w_0)$. Next, a $\Uparrow (r_0 w_1 r_1 w_0)$ march is performed. In this march, each memory cell is first read and verified to be the expected value 0. The cell is then written with the complementary value 1, read

again with the expected value being 1, and finally written back with the original value 0. This march operation is shown as March 1 in Figure 13.

Once the first march is completed, the memory has to be loaded according to the next background. In this case, the next background is the second column in the background matrix. This background change is achieved using a minimum number of operations by comparing the current background with the next background as follows:

- if the P bit of a cell is the same as the N bit, no operation is performed on that cell. The address is incremented to the next memory cell.
- if the P bit of a cell is different from the N bit, the cell is read with the expected value equal to P, and then written with the complementary value $\overline{P}$ (which is also equal to N). As with the first case, the address is then incremented to the next cell.

These steps are applied to all the memory cells in ascending address order. (Any other arbitrary address order could in fact be used.) In the example in Figure 13, since the P bit and the N bit for cell 0 are both equal to 0 (see Figure 12), no read or write operations are required. The P bit and the N bit for cell 1 however, are not the same. In this case, cell 1 is read with the expected value $P = 0$, and written with the complement value $\overline{P} = 1$. Cells 2 and 3 also undergo the same transition.

At the end of the background change, the background number is incremented to point to the second column so that the second background is now the current background. A $\Uparrow (r_P w_{\overline{P}} r_{\overline{P}} w_P)$ march is then applied to memory in ascending order, where the P bits now correspond to the bits of the second background. This march operation is labeled as March 2 in Figure 13. These march and background change steps are repeated until all m backgrounds have been applied to the RAM.

At the end of the test, all the cells are read again, with the expected values equal to the P bits of the last column in the background matrix.

Pseudo-code for Cockburn's algorithm is given in Figure 14.

```
const n; /* number of RAM cells */
const m; /* number of backgrounds in background matrix */
const BGM[n,m]; /* n × m background matrix */
type address = 0 ... (n-1); bit = (0,1); bg_position = 0, ... m − 1
var a: address; bg_num: bg_position; P, N : bit; last_dbgr : boolean;
begin
    /* load first background into RAM */
    for a := 0 to n − 1 do
        P := BGM[a,0]; w_P^a;
    endfor;
    /* consider each background */
    bg_num := 0;
    last_dbgr := false;
    repeat
        /* ascending r_P w_P r_P w_P march */
        for a := 0 to n − 1 do
            P := BGM[a,bg_num]; r_P^a w_P^a r_P^a w_P^a;
        endfor;
        if bg_num = m − 1 then
            /* have reached last background */
            last_dbgr := true;
        else
            /* load next background into RAM */
            for a := 0 to n − 1 do
                P := BGM[a,bg_num];
                N := BGM[a,bg_num + 1];
                if P ≠ N then r_P^a; w_P^a; endif;
            endfor;
            bg_num := bg_num + 1;
        endif;
    until last_dbgr;
    /* read each cell one last time */
    for a := 0 to n − 1 do
        P := BGM[a,m − 1]; r_P^a;
    endfor;
end;
```

Figure 14: Deterministic Test Algorithm

A central theorem, which is proved in [22], is as follows:

Theorem 1: Cockburn's test detects all single V-coupling faults if the underlying code is $(n, V-1)$-exhaustive.

Proof: Refer to [22] for the full proof. Informally, the argument has two steps.

First, the fact that the background matrix is an $(n, V-1)$-exhaustive code guarantees that any single V-coupling fault will be excited at least once. Therefore, if a fault is present, an error is guaranteed to appear at least once. The second step shows that if an error is produced, it is guaranteed to be detected by a read operation. The theorem follows from the two steps. □

The properties of an $(n, V-1)$-exhaustive code will be discussed in the following section.

3.2 $(n, V-1)$-exhaustive Codes

Consider an $n \times m$ binary matrix A. The rows of A form an $(n, V-1)$-exhaustive code if:

1. $V - 1 \leq n$, and
2. for all possible projections of matrix A onto $V-1$ rows, all $2^{(V-1)}$ possible binary $(V-1)$-tuples appear as column vectors.

3.2.1 $(n, 2)$-exhaustive Codes

Figure 15 shows an example of an $(8, 2)$-exhaustive code [22].

This code can be divided into four fields: the first field is an all-zero column. The second field is a $\log_2(n)$-bit wide binary count-up sequence from 0 to $n-1$. In the $(8, 2)$-exhaustive code example in Figure 15, the count-up sequence is $\log_2(8) = 3$ bits wide, and goes from 000 up to 111. The third field is an all-one column. The last field consists of the binary representation of the number of zeros in the second field. In Figure 15, the entry in the fourth field for the first row is 11 corresponding to the three zeros in the binary number 000. Similarly, the entry for the second row is 10 because the binary number 001 in the second field contains two zeros.

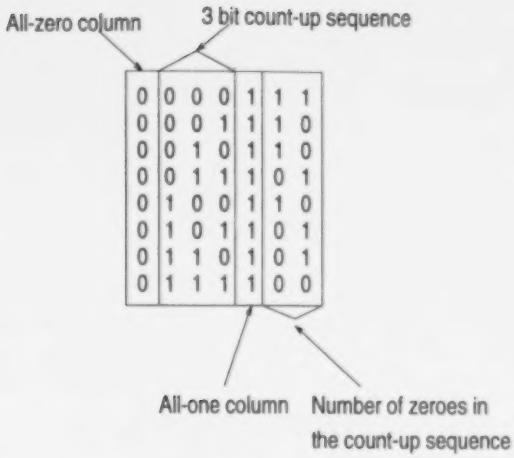

All-zero column
3 bit count-up sequence
0 0 0 0 1 1 1
0 0 0 1 1 1 0
0 0 1 0 1 1 0
0 0 1 1 1 0 1
0 1 0 0 1 1 0
0 1 0 1 1 0 1
0 1 1 0 1 0 1
0 1 1 1 1 0 0
All-one column
Number of zeroes in
the count-up sequence

Figure 15: $(8,2)$-exhaustive code

The length of the code is the total number of bits in each row. In this example, the length of the $(8,2)$-exhaustive code is 7. The following proposition is readily verified from the code definition.

Proposition 1 The length of an $(n,2)$-exhaustive code with the structure described is equal to $2 + \lceil \log_2 n \rceil + \lceil \log_2(\lceil \log_2 n \rceil + 1) \rceil$.

Proposition 2: The code illustrated in Figure 15 is $(n,2)$-exhaustive.

Proof: We have to show that for any two rows in the matrix, all 2^2 possible binary 2-tuples 00, 01, 10 and 11 appears as column vectors. The first field guarantees that any two rows will have the 00 tuple. Similarly, the third field provides the 11 tuple. To show that any two rows will have both the 01 and 10 tuples, consider two arbitrary rows x and y, where $x < y$. Let the binary number b_x be the entry in the second field for x, and let b_y be the corresponding entry for y. Since $b_x < b_y$, there must exist a bit position i in b_x and b_y where $b_x(i) = 0$ and $b_y(i) = 1$; otherwise we reach a contradiction with the assumption that $x < y$.

To show that any two rows will have the 10 tuple, let w_x be the entry in the fourth field for x, and let w_y be the corresponding entry for y. We have to consider three cases:

Case 1: $(w_x > w_y)$ Since $w_x > w_y$, there must exist a bit position j where $w_x(j) = 1$ and $w_y(j) = 0$.

Case 2: $(w_x = w_y)$ Since $w_x = w_y$, b_x contains the same number of ones and zeros as b_y. Since $b_x \neq b_y$, b_y cannot have ones in exactly the same bit positions where b_x has ones. Therefore, there must be at least one bit position j where $b_x(j) = 1$ and $b_y(j) = 0$.

Case 3: $(w_x < w_y)$ Since $w_x < w_y$, there are more zeros in b_y than in b_x. Therefore, there must be at least one bit position j where $b_x(j) = 1$ and $b_y(j) = 0$. □

In order to show that this code will detect all single 3-coupling faults in an $n \times 1$ RAM, we will enumerate all the possible 3-coupling faults that can occur. Table 2 lists all the 3-coupling faults that can occur involving three arbitrary cells, c_1, c_2 and c_3. Fault $1a$ refers to an idempotent 3-coupling fault where a write operation that changes the content of c_1 from 0 to 1 (denoted by $\uparrow c_1$) also forces c_2 to 1, when c_3 contains a 0. Fault $1b$ is similar to Fault $1a$ except that c_3 contains a 1 thus an error can occur in c_2 only if $c_3 = 1$. Faults $2a$ and $2b$ are idempotent 3-coupling faults where a 0 to 1 transition in c_1 forces c_2 to 0. The analogous 3-coupling faults that are caused by $\downarrow c_1$ are Faults $3a$, $3b$, $4a$ and $4b$.

To detect all the single 3-coupling faults in Table 2, the rows of the background matrix used in the deterministic test described in Section 3.1 will have to form an $(n, 2)$-exhaustive code. This means that any two rows out of n rows in the matrix will have all four possible 2-bit binary patterns as column vectors. Since

Fault 1a	$\uparrow c_1 \Rightarrow \uparrow c_2, c_3 = 0$
Fault 1b	$\uparrow c_1 \Rightarrow \uparrow c_2, c_3 = 1$
Fault 2a	$\uparrow c_1 \Rightarrow \downarrow c_2, c_3 = 0$
Fault 2b	$\uparrow c_1 \Rightarrow \downarrow c_2, c_3 = 1$
Fault 3a	$\downarrow c_1 \Rightarrow \uparrow c_2, c_3 = 0$
Fault 3b	$\downarrow c_1 \Rightarrow \uparrow c_2, c_3 = 1$
Fault 4a	$\downarrow c_1 \Rightarrow \downarrow c_2, c_3 = 0$
Fault 4b	$\downarrow c_1 \Rightarrow \downarrow c_2, c_3 = 1$

Table 2: All Possible 3-coupling Faults Between Three Cells c_1, c_2 and c_3.

all the columns in the matrix are test vectors that are loaded into the RAM under test, any two cells in the $n \times 1$ RAM will go through all the possible (2^2) joint states (00,01,10,11). Referring to Table 2, the two cells that are relevant here are those denoted by c_2 and c_3. Two conditions must be satisfied for an error to be induced in cell c_2: (1) Cells c_2 and c_3 must contain two particular values; (2) When cells c_2 and c_3 contains those values, cell c_1 is written in the appropriate direction and the contents of c_1 are changed. Condition (1) is met in the test because an $(n, 2)$-exhaustive code guarantees that cells c_2 and c_3 will go through all possible states, which means sometime during the test, the cells will contain the "right" values at least once. Condition (2) is satisfied because a $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march is performed for each background, which means that when cells c_2 and c_3 contain the required values, cell c_1 is guaranteed to undergo both $\uparrow$ and $\downarrow$, one of which is the appropriate transition that triggers the fault. Note that the states that (c_2, c_3) goes through do not necessarily have to be in the order 00, 01, 10, 11.

3.2.2 $(n,3)$-exhaustive Codes

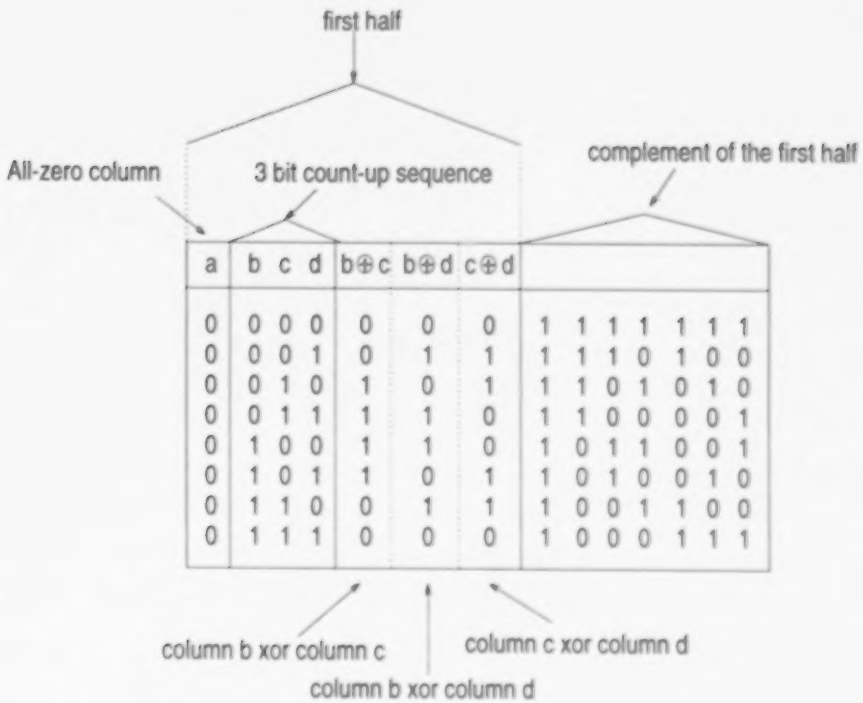

first half
All-zero column
3 bit count-up sequence
complement of the first half
a b c d b⊕c b⊕d c⊕d
0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 1 0 1 1 1 1 1 0 1 0 0
0 0 1 0 1 0 1 1 1 0 1 0 1 0
0 0 1 1 1 1 0 1 1 0 0 0 0 1
0 1 0 0 1 1 0 1 0 1 1 0 0 1
0 1 0 1 1 0 1 1 0 1 0 0 1 0
0 1 1 0 0 1 1 1 0 0 1 1 0 0
0 1 1 1 0 0 0 1 0 0 0 1 1 1
column b xor column c
column b xor column d
column c xor column d

Figure 16: (8,3)-exhaustive code

An easily-generated (8,3)-exhaustive code is shown in Figure 16. This type of $(n,3)$-exhaustive code is divided into two halves. There are three fields in the first half. Similar to the (n,2)-exhaustive code, the first field is an all-zero column and the second field is a $\log_2(n)$-bits wide binary ascending sequence. The third field is obtained by performing exclusive-OR operations on all possible combinations of two bit positions from the second field. In the example in Figure 16, there are three bits (or columns) in the second field labeled b, c and d. In this case, there are ${}^3C_2 = \frac{3!}{2!1!} = 3$ combinations of two rows that can be XOR-ed together; namely, $b \oplus c$, $b \oplus d$ and $c \oplus d$. The second half of the code is obtained by taking the logical

complement of all the entries in the first half.

The following proposition is readily verified from the code definition.

Proposition 3: The length of an $(n,3)$-exhaustive code with the structure described above is equal to $2 + \lceil \log_2 n \rceil + \lceil \log_2 n \rceil^2$.

Proposition 4: The code illustrated in Figure 16 is $(n,3)$-exhaustive.

Proof: To prove this proposition, we must prove that any three arbitrary rows from the code will have $2^3 = 8$ distinct 3-bit tuples. Let b_x, b_y and b_z be the corresponding entries in the second field for three rows x, y, and z. The first column guarantees one distinct tuple, i.e. the 000 tuple. Next, we will prove that the second field has at least two distinct tuples, $b_x(i)b_y(i)b_z(i)$ and $b_x(j)b_y(j)b_z(j)$, where i and j are two distinct bit positions in b_x, b_y and b_z, and $b_x(i)b_y(i)b_z(i) \neq \overline{b_x(j)b_y(j)b_z(j)}$.

Since $b_x \neq b_y$, there must be a position i where $b_x(i) \neq b_y(i)$. Let T be the 3-bit tuple $b_x(i)b_y(i)b_z(i)$. Since $b_x(i) \neq b_y(i)$, we can write $T = a\bar{a}x_a$ where $a \in \{0,1\}$ and $x_a \in \{0,1\}$. There are two cases to consider:

Case 1: $(x_a = a)$ The tuple T has the form $T_1 = a\bar{a}a$. In this case, $b_x(i) = b_z(i)$. Since $b_x \neq b_z$, there must be at least a second bit position j where $b_x(j) \neq b_z(j)$. Let T_2 be the tuple $b_x(j)b_y(j)b_z(j)$. Therefore $T_2 = bx_b\bar{b}$ where $b \in \{0,1\}$ and $x_b \in \{0,1\}$. To prove that T_1 and T_2 are distinct, assume that $T_1 = T_2$. Then, $a = b$, $\bar{a} = x_b$, and $a = \bar{b}$ which contradicts $a = b$. Therefore T_1 and T_2 are distinct. To show that $T_1 \neq \overline{T_2}$, assume that $T_1 = \overline{T_2}$, then $a = \bar{b}$, $\bar{a} = \bar{x}_b$ and $a = b$ which contradicts $a = \bar{b}$. Therefore $T_1 \neq \overline{T_2}$.

Case 2: $(x_a = \bar{a})$ The tuple T has the form $T_1 = a\bar{a}\bar{a}$. In this case $b_y(i) = b_z(i)$. Since $b_y \neq b_z$, there must be at least a second bit position j where

$b_y(j) \neq b_z(j)$. Let T_2 be the tuple $b_x(j)b_y(j)b_z(j)$. Therefore $T_2 = x_b b\bar{b}$ where $b \in \{0,1\}$ and $x_b \in \{0,1\}$. To prove that T_1 and T_2 are distinct, assume that $T_1 = T_2$. Then, $a = x_b$, $\bar{a} = b$, and $\bar{a} = \bar{b}$ which contradicts $\bar{a} = b$. Therefore T_1 and T_2 are distinct. To show that $T_1 \neq \overline{T_2}$, assume that $T_1 = \overline{T_2}$, then $a = \bar{x}_b$, $\bar{a} = \bar{b}$ and $\bar{a} = b$ which contradicts $\bar{a} = \bar{b}$. Therefore $T_1 \neq \overline{T_2}$.

Clearly in Case 1, $T_1 = a\bar{a}a$ and $T_2 = bx_b\bar{b}$ cannot be either 000 or 111. Similarly, in Case 2, $T_1 = a\bar{a}\bar{a}$ and $T_2 = x_b b\bar{b}$ cannot be equal to 000 or 111.

At this point, we have three distinct tuples, one from the first field and two tuples from the second field. We now show that using the two distinct tuples from the second field, we obtain a third distinct tuple in the third field.

Let $T_{T_1 \oplus T_2}$ be the tuple resulting from $T_1 \oplus T_2$, where $\oplus$ denotes bit-wise exclusive-OR. Assume that $T_{T_1 \oplus T_2} = T_1$. Then $T_2 = 000$ which contradicts the above result that $T_2 \neq 000$. Therefore $T_{T_1 \oplus T_2} \neq T_1$. Assume that $T_{T_1 \oplus T_2} = \overline{T_1}$. Then $T_2 = 111$ which contradicts the above result that $T_2 \neq 111$. Therefore $T_{T_1 \oplus T_2} \neq \overline{T_1}$.

Assume that $T_{T_1 \oplus T_2} = T_2$. Then $T_1 = 000$ which contradicts the above result that $T_1 \neq 000$. Therefore $T_{T_1 \oplus T_2} \neq T_2$. Assume that $T_{T_1 \oplus T_2} = \overline{T_2}$. Then $T_1 = 111$ which contradicts the above result that $T_1 \neq 111$. Therefore $T_{T_1 \oplus T_2} \neq \overline{T_2}$.

Next, we will prove that $T_{T_1 \oplus T_2} \neq 000$ or 111. Assume that $T_{T_1 \oplus T_2} = 000$. Then $T_1 = T_2$ which contradicts the above result that $T_1 \neq T_2$. Therefore $T_{T_1 \oplus T_2} \neq 000$. Assume that $T_{T_1 \oplus T_2} = 111$. Then $T_1 = \overline{T_2}$ which contradicts the above result that $T_1 \neq \overline{T_2}$.

We have now shown that for any three rows x, y, and z, the first field contains

000, the second field contains two more distinct tuples, and the third contains a fourth distinct tuple. Furthermore, none of these tuples is the complement of the other three. Using the four distinct tuples in the first half of the code, we obtain four more tuples in the second half. Since we have shown that no two distinct tuples in the first half form a complementary pair, by taking the complement of the first half, four additional distinct tuples are obtained. Therefore, any rows from the code in Figure 16 will have 2^3 distinct 3-bit tuples, which proves that the code is $(n,3)$-exhaustive. □

3.2.3 $(n,4)$-exhaustive Codes for T-neighborhoods

In section 2.3, scrambled physical neighborhood pattern-sensitive faults (*PNPSFs*) were discussed. It was observed that the effects of row and column address scrambling on the PNPSFs are as follows:

- The victim or base cell B can be displaced anywhere in the memory map;
- The N and S neighbors can be displaced anywhere along the same column as B;
- The E and W neighbors can be displaced anywhere along the same row as B.

An *active* scrambled PNPSF is said to be present if a transition in an *aggressor* cell i (i can be N, E, W or S) causes the *victim* cell j ($j = B$) to change to an erroneous state when particular values are stored in the remaining cells (which we will label as k_1, k_2 and k_3) in the neighborhood. The deterministic test described in Section 3.1 will detect all single active scrambled PNPSFs if the underlying background matrix is $(n,4)$-exhaustive with respect to scrambled T-neighborhoods. A *T-neighborhood* is a set of four cells (j, k_1, k_2 and k_3) that correspond to a

standard PNPSF neighborhood. A *scrambled T-neighborhood* is a T-neighborhood in which the cells have been scrambled. An example of the logical neighborhood of a scrambled PNPSF is shown in Figure 17(i). The aggressor cell i is in the same column as the victim cell j. The corresponding scrambled T-neighborhood is shown in Figure 17(ii). The cells k_1 and k_3 are in the same row as j while cell k_2 is in the same column as j.

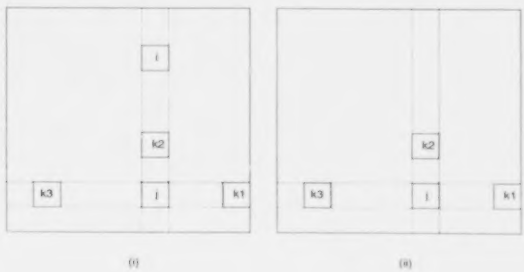


Figure 17: (i) Logical Neighborhood for a Scrambled Pattern-Sensitive Fault; (ii) Corresponding Scrambled T-neighborhood

We assume that the storage cells in the $n \times 1$ RAM are arranged in a square $\sqrt{n} \times \sqrt{n}$ grid.

The following construction is due to Cockburn.

To construct an $(n, 4)$-exhaustive code with respect to scrambled T-neighborhoods, a $(\sqrt{n}, 3)$-exhaustive base matrix is used. The structure of the $(\sqrt{n}, 3)$-exhaustive code is as described in the previous subsection. Only the first half of the $(\sqrt{n}, 3)$-exhaustive code is used. The fields in this code are labeled Zone I, Zone II and Zone III as shown in Figure 18.

In this example, $n = 16$ and thus the base matrix should be (4,3)-exhaustive. Zone I refers to the all-zero column. Zone II is the ascending binary sequence. Zone III is obtained by XOR-ing all pairs of columns from Zone II. Two column pointers, $col1$ and $col2$, are used in the construction. We define the "product" of

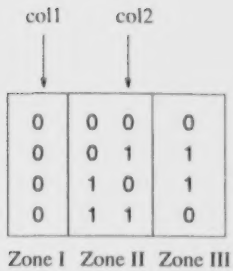


Figure 18: (4,3)-exhaustive Base Matrix Used in the Construction of a $(16, 4)$-exhaustive Code with Respect to Scrambled T-neighborhoods

two column vectors, $col1 \times col2$, as a matrix M where each entry of M, $M(i, j)$ is given by $col1(i) \oplus col2(j)$. The entries of such a matrix M, when read row by row, will form the columns of the new code, as we describe next. The columns of the $(n, 4)$-exhaustive code with respect to scrambled T-neighborhoods are constructed as follows:

Code Construction 1:

Step 1:

for $col1$ = first column **to** last column in Zone II **do**

for $col2 = col1 + 1$ **to** last column in Zone III **do**

$M = col1 \times col2$ with col1 in vertical direction and

col2 in horizontal direction

end

end

Step 2: Repeat Step 1 with $col1$ and $col2$ interchanged.

Step 3: Complement all the entries in the matrices obtained in Step 1 and Step 2.

Figure 19 shows Step 1 using the base matrix in Figure 18. In this step, $col1$ is initialized to the first column in the base matrix while $col2$ is initialized to the

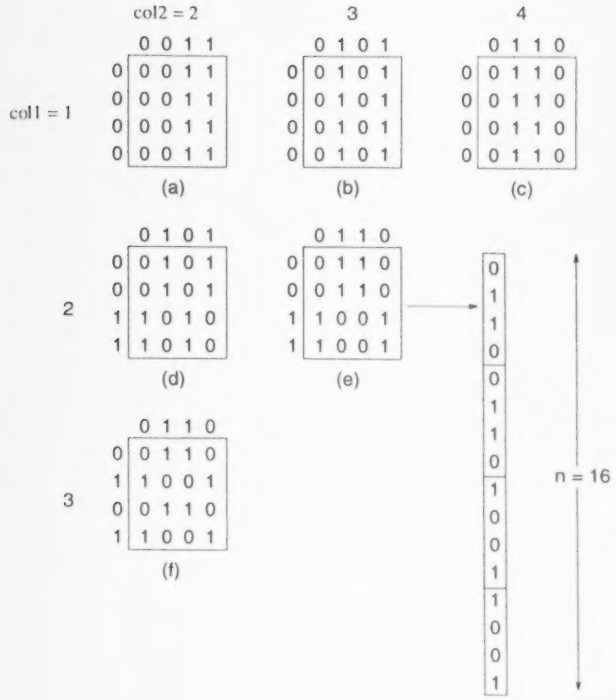


Figure 19: Construction of a $(16, 4)$-exhaustive Code With Respect to Scrambled T-neighborhoods (Step 1)

second column. The first resulting matrix M is shown in Figure 19(a). Figures 19(b), (c), (d), (e) and (f) shows the remaining XOR-matrices obtained by Step 1. Each matrix corresponds to a background (column) in the new code. For example, the rows in the XOR-matrix in Figure 19(e) can be concatenated together to obtain the fifth 16-bit background in the new code.

In Step 2, Step 1 is repeated with *col*1 and *col*2 are interchanged. The XOR-matrices obtained in this step are the same $\sqrt{n} \times \sqrt{n}$ matrices obtained in Step 1 transposed about the main diagonal. Note that the product of the two vectors is still performed with col1 in the vertical direction and col2 in the horizontal direction. In Step 3, the entries of the matrices obtained in Step 1 and Step 2

are complemented. The resulting background matrix for our example is shown in Figure 20.

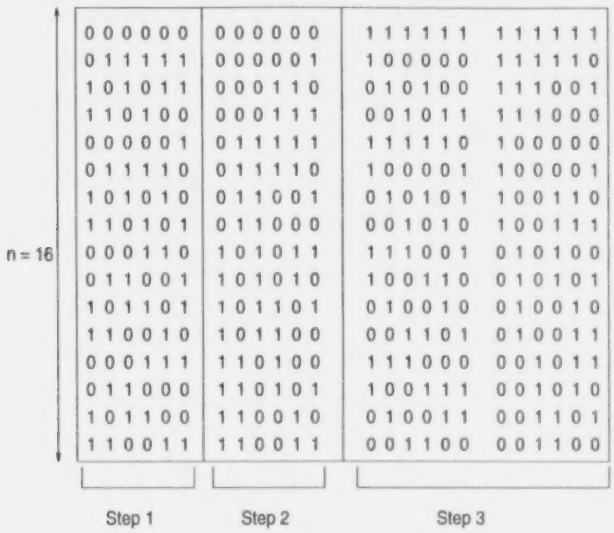


Figure 20: $(16, 4)$-exhaustive Code With Respect to Scrambled T-neighborhoods

Proposition 5: Given an $(n, 4)$-exhaustive code, Construction 1 produces a code of length $2\lceil \log_2 \sqrt{n} \rceil^2 (1 + \lceil \log_2 \sqrt{n} \rceil)$.

Proof: Each matrix corresponds to a background in the constructed code. The length of the code is thus equal to the number of matrices that are generated. Consider the number of matrices generated in Step 1. Zone I is one bit wide. Zone II is $\lceil \log_2 \sqrt{n} \rceil$-bit wide. Therefore the outer for loop goes from $col1 = 1$ to $col1 = 1 + \lceil \log_2 \sqrt{n} \rceil$. Zone III has a length of $\frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$. Therefore, the inner for loop goes from $col2 = col1 + 1$ to $col2 = 1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$. Let $A = 1 + \lceil \log_2 \sqrt{n} \rceil$ and $B = 1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$. The number L_1 of matrices that

are generated in Step I can be written as follows:

$$\begin{aligned} L_1 &= \sum_{col1=1}^{A} \left(\sum_{col2=col1+1}^{B} 1 \right) \\ &= \sum_{col1=1}^{A} (B - col1) \\ &= \sum_{col1=1}^{A} B - \sum_{col1=1}^{A} col1 \\ &= AB - \frac{A(A+1)}{2} \\ &= A(B - \frac{A+1}{2}) \end{aligned}$$

The number of matrices generated in Step 2 is the same as L_1. The number of matrices generated in Step 3 is $2 \times L_1$. Therefore, the total length L of the code is equal to $L = 4 \times L_1$ Substituting $A = 1 + \lceil \log_2 \sqrt{n} \rceil$ and $B = 1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$ into L_1, we get:

$$\begin{aligned} L &= 4(1 + \lceil \log_2 \sqrt{n} \rceil) \left(1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2} - \frac{\lceil \log_2 \sqrt{n} \rceil + 2}{2}\right) \\ &= 2\lceil \log_2 \sqrt{n} \rceil^2 (1 + \lceil \log_2 \sqrt{n} \rceil) \quad \square \end{aligned}$$

For simplicity we will prove that construction 1 produces $(n, 4)$-exhaustive codes with respect to all scrambled T-neighborhoods in two steps. We will first modify construction 1 so that $col2$ starts from $col1$ instead of $col1 + 1$ in the inner for loop, and then prove that this new construction produces $(n, 4)$-exhaustive codes with respect to all scrambled T-neighborhood (Proposition 6). The modified construction contains all the same backgrounds as the original construction, plus some additional backgrounds. Afterwards we will prove that if $col2$ starts from $col1 + 1$, the codes produced are still $(n, 4)$-exhaustive with respect to all scrambled T-neighborhood (Proposition 7).

Proposition 6: The codes produced by a modified construction 1 (where $col2$ starts from $col1$ instead of $col1+1$ in the inner for loop) are $(n,4)$-exhaustive with respect to all scrambled T-neighborhoods.

Proof: Consider an arbitrary scrambled T-neighborhood.

There are two cases to consider:

Case 1: The aggressor cell S is in the same column as the victim cell B. (We have chosen cell S to denote the aggressor cell. The following argument is independent of whether the aggressor cell is cell N or cell S.) Let cells W, B and E be in columns a, b and c, respectively, where a, b and c are three distinct values in the range of $0,\ldots,\sqrt{n}-1$. We assume that the cell array is a $\sqrt{n}\times\sqrt{n}$ grid. Also let cells N and B be in rows d and e, respectively, where d and e are two distinct values in the range $0,\ldots,\sqrt{n}-1$.

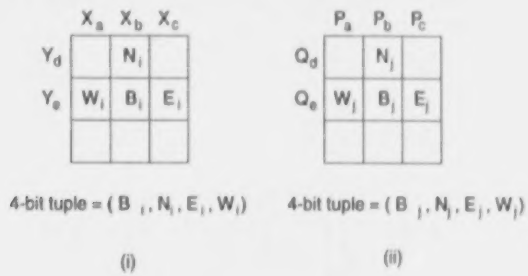


Figure 21: Scrambled Physical T-Neighborhoods

Consider two arbitrary backgrounds i and j where i and j are two distinct values in the range $0,\ldots,m-1$. The scrambled T-neighborhoods after applying backgrounds i and j to the memory are shown in Figure 21. (In this figure we have shown the neighborhoods without scrambling. This is done for simplicity. The following argument is independent of the scrambling that may be present).

Using Proposition 2, any two arbitrary rows from Zone I and Zone II of the base matrix have at least two distinct 2-bit tuples: the 00 tuple and the 01 tuple. Using Proposition 4, any three arbitrary rows from the $(\sqrt{n}, 3)$-exhaustive base matrix have at least four distinct 3-bit tuples and none of these tuples is the complement of the other three tuples. In Step 1 of code construction 1, the product of the two vectors *col*1 and *col*2 is performed with *col*1 in the vertical direction and *col*2 in the horizontal direction. Therefore the 2-bit vector $col1(d)col1(e)$ will go through at least two distinct states. Similarly and independently, the 3-bit vector $col2(a)col2(b)col2(c)$ will go through at least four distinct states.

Let S_1 be the set that contains the two distinct states for $col1(d)col1(e)$. Let S_2 be the set that contains the four distinct states for $col2(a)col2(b)col2(c)$. Let $X = X_a X_b X_c$ be the value for the 3-bit vector $col2(a)col2(b)col2(c)$ for background i and $X \in S_2$. Let $Y = Y_d Y_e$ be the state for the 2-bit vector $col1(d)col1(e)$ for background i and $Y \in S_1$. Similarly, let $P = P_a P_b P_c$ be the value of the 3-bit vector $col1(a)col1(b)col1(c)$ for background j and $P \in S_2$. Lastly, let $Q = Q_d Q_e$ be the value for the 2-bit vector $col1(d)col1(e)$ for background j and $Q \in S_1$.

Let i identify the background which must exist such that $Y = 00$. Similarly, let j identify the background which must exist such that $Q = 01$. Variables d and e are chosen, without loss of generality, such that $Q_d = 0$ and $Q_e = 1$. We will prove that the 4-bit tuple $B_i N_i E_i W_i$, obtained from the product $X \times Y$ of the two column vectors X and Y, is distinct from the 4-bit tuple $B_j N_j E_j W_j$ which is obtained from $P \times Q$. Furthermore, we will show that $B_i N_i E_i W_i \neq \overline{B_j N_j E_j W_j}$.

Assume for a moment that $B_iN_iE_iW_i = B_jN_jE_jW_j$. Then,

$$Y_d \oplus X_b = Q_d \oplus P_b \quad (1)$$

$$Y_e \oplus X_b = Q_e \oplus P_b \quad (2)$$

$$Y_e \oplus X_a = Q_e \oplus P_a \quad (3)$$

$$Y_e \oplus X_c = Q_e \oplus P_c \quad (4)$$

Assume that $X_b \neq P_b$ implying that $P_b = \overline{X_b}$. From (1) and (2), $Y_d = \overline{Q_d}$ and $Y_e = \overline{Q_e}$ which is a contradiction since $Y_dY_e = 00 \neq \overline{Q_dQ_e} = \overline{01}$. Therefore, $X_b = P_b$. Using this result and equations (1) and (2) $Y_d = Q_d$ and $Y_e = Q_e$. However, this is a contradiction since $Y_dY_e \neq Q_dQ_e$. Therefore, we conclude that $B_iN_iE_iW_i \neq B_jN_jE_jW_j$.

Next we will show that $B_iN_iE_iW_i \neq \overline{B_jN_jE_jW_j}$. Assume for a moment that $B_iN_iE_iW_i = \overline{B_jN_jE_jW_j}$. Then,

$$Y_d \oplus X_b = \overline{Q_d \oplus P_b} \quad (5)$$

$$Y_e \oplus X_b = \overline{Q_e \oplus P_b} \quad (6)$$

$$Y_e \oplus X_a = \overline{Q_e \oplus P_a} \quad (7)$$

$$Y_e \oplus X_c = \overline{Q_e \oplus P_c} \quad (8)$$

Assume that $X_b \neq P_b$ implying $P_b = \overline{X_b}$. From (5) and (6), $Y_d = Q_d$ and $Y_e = Q_e$ which is a contradiction since $Y_dY_e \neq Q_dQ_e$. Therefore, $X_b = P_b$. Using this result and equations (5) and (6), $Y_d = \overline{Q_d}$ and $Y_e = \overline{Q_e}$. However, this is a contradiction since $Y_dY_e \neq \overline{Q_dQ_e}$. Therefore, we conclude that $B_iN_iE_iW_i \neq \overline{B_jN_jE_jW_j}$. Using the similar arguments, it can be shown that the 4-bit tuple obtained from $X \times Y$ is also distinct from the tuple obtained from $P \times Q$ where $X \neq P$ and $X \neq \overline{P}$.

We have proved that the 4-bit tuple obtained from the product of two vectors, $X \times Y$ is distinct from the 4-bit tuple obtained from the product of two vectors, $P \times Q$ when at least one member in the product is distinct from the corresponding member in the second product, i.e. either $Y \neq Q$ and $Y \neq \overline{Q}$, or $X \neq P$ or $X \neq \overline{P}$. We have also shown that the 2-bit vector $col1(d)col1(e)$ will go through at least the 00 and 01 states and that the 3-bit vector $col2(a)col2(b)col2(c)$ will go through at least four distinct states. Therefore, in Step 1 of the code construction, any 4-cell scrambled T-neighborhoods where the aggressor cell is in the same column as B, will go through $4 \times 2 = 8$ distinct states.

Case 2: The aggressor cell is in the same row as the victim cell B.
In Step 2, $col1$ and $col2$ are interchanged. Using similar arguments as the ones in Case 1, any 3-bit tuple from $col1$ will go through at least four distinct states and any 2-bit tuple from $col2$ will go through at least two distinct states. Therefore, any scrambled neighborhood where the aggressor cell is in the same row as the victim cell B will go through at least eight distinct states.

In Step 1, we omit the matrices obtained when $col2 < col1$ since they will be generated in Step 2. For example, the matrix $M = col1 \times col2$ obtained when $col1$ = second column and $col2$ = first column in Step 2 is the same as the matrix $M = col1 \times col2$ obtained when $col2$ = first column and $col1$=second column. Therefore, the col2 pointer does not have to start from the first column. It can start from $col2 = col1$.

We have shown that in both cases, any 4-cell scrambled T-neighborhoods will go through at least eight distinct states and none of these states is the complement of the remaining seven states. Therefore in Step 3, when we

take the complement of all the entries in the XOR-matrices that are found in Step 1, the 4-cell scrambled T-neighborhoods will go through the remaining 8 states. Therefore any 4-cell scrambled T neighborhood will go through 2^4 states, which proves that the codes generated by the modified construction 1 are $(n, 4)$-exhaustive with respect to scrambled T-neighborhoods. □

Proposition 7: Code construction 1 produces $(n, 4)$-exhaustive codes with respect to all scrambled T-neighborhoods .

Proof: Consider an arbitrary scrambled T-neighborhood. We first consider the case where the triggering cell (S) is in the same column as the base cell B. Let cells N and B be in rows d and e, where d and e are two distinct values in the range $0, \ldots, \sqrt{n} - 1$. We assume that the cell array is a $\sqrt{n} \times \sqrt{n}$ grid.

To prove that code construction 1 produces $(n, 4)$-exhaustive codes with respect to all scrambled T-neighborhoods, we have to show that by removing the XOR matrices M obtained when $col1 = col2$, we do not remove the only instance of any of the 2^4 possible states for the 4-cell scrambled T-neighborhoods. Let P denote $col1$ and $col2$ when they are both identical. Consider $M_{PP} = P \times P$. It must be true that P is a column in Zones I or II of the base matrix. Assume that a particular $NWBE$ neighborhood occurs in M_{PP}. It can be seen from Figure 22 that the same neighborhood will occur in at least one other background or matrix $M_{PQ} = Q \times P$ if the 2-bit tuple $P(d)P(e)$ is equal to the 2-bit tuple $Q(d)Q(e)$. We will prove that there exists at least one other column Q where $P \neq Q$ and $P(d)P(e) = Q(d)Q(e)$.

Since P is from Zones I or Zones II (i.e. from a $(\sqrt{n}, 2)$-exhaustive code), then it must be true that Q is from Zones I, II or III (i.e. from a $(\sqrt{n}, 3)$-exhaustive code) in either Step 1 or Step 2 of construction 1. Let "a" be an

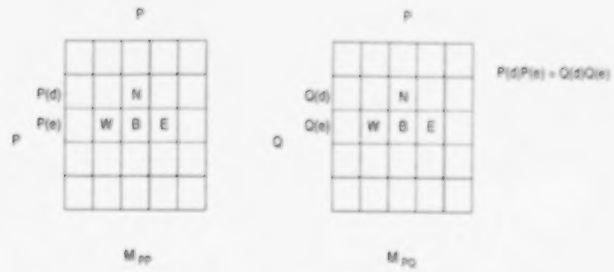


Figure 22: Backgrounds $M_{PP} = P \times P$ and $M_{PQ} = P \times Q$

appropriate boolean value. Therefore, both of the 3-bit patterns $aP(d)P(e)$ and $\bar{a}P(d)P(e)$ must be present in Zones I, II or III, where a can occur in any of the remaining $\sqrt{n} - 2$ rows in the XOR-matrices other than rows d and e. Since both patterns cannot be present in P alone, then we conclude that there must be one other column Q where $P \neq Q$ and $P(d)P(e) = Q(d)Q(e)$. We have thus shown that there exists a neighborhood in M_{PQ} which is identical to the neighborhood in M_{PP}, and $P \neq Q$.

Due to the symmetry of the constructed matrices about the main diagonal, the same arguments can be used for the case when the triggering cell is in the same row as B.

We have now shown that the scrambled T-neighborhoods that are present in the matrices $M = col1 \times col2$, where $col1 = col2$, are guaranteed to be present in at least one other background where $col1 \neq col2$. Therefore, by removing the XOR matrices M obtained from $col1 \times col2$ when $col1 = col2$, we do not remove any of the 2^4 possible states for the 4-cell scrambled T-neighborhoods, and hence code construction 1 produces $(n, 4)$-exhaustive codes with respect to all scrambled T-neighborhoods. □

3.2.4 $(n,4)$-exhaustive Codes for Neighborhoods of Type 1

A *passive* scrambled PNPSF is said to be present if the content of the victim cell j ($j = B$) cannot be changed by write operations addressed to cell j when a particular pattern of values is stored in the N, E, W and S cells in the scrambled neighborhood [23]. A *static* scrambled PNPSF is said to be present if the state of the victim cell B is forced to a particular value, 0 or 1, when a particular pattern of values is present in the N, E, W and S cells in the scrambled neighborhood [23]. The deterministic test described in Section 3.1 will detect all single static and passive scrambled PNPSFs if the underlying background matrix is $(n,4)$-exhaustive with respect to *neighborhoods of type 1*. A neighborhood of type 1 is a set of four cells (N, E, W and S) that correspond to a standard PNPSF neighborhood. A *scrambled neighborhood of type 1* is a neighborhood of type 1 in which all the cells of the neighborhood have been scrambled. An example of the logical neighborhood of a scrambled PNPSF is shown in Figure 23(i). The victim cell B has two neighbors in the same column, and two neighbors in the same row. The corresponding scrambled neighborhood of type 1 is shown in Figure 23(ii).

Figure 23: (i) Logical Neighborhood for Scrambled Pattern-Sensitive Faults; (ii) Corresponding Scrambled Neighborhood of Type 1

The storage cells in the $n \times 1$ RAM are again assumed to be arranged in a square $\sqrt{n} \times \sqrt{n}$ grid.

Similar to the codes for scrambled T-neighborhoods, a $(\sqrt{n},3)$-exhaustive base matrix is used to construct an $(n,4)$-exhaustive code with respect to scrambled neighborhoods of type 1. This construction is also due to Cockburn. The fields in the base matrix are again labeled Zone I, Zone II and Zone III (see Figure 18). The construction process is as follows:

Code Construction 2:

Step 1:

for $col1$ = first column **to** last column in Zone III **do**

for $col2$ = first column **to** last column in Zone III **do**

$M = col1 \times col2$ with col1 in vertical direction

and col2 in horizontal direction

end

end

Step 2: Complement all entries in the XOR-matrices found in Step 1.

Figure 24 shows the XOR matrices for Step 1 obtained from using the base matrix in Figure 18. In this step, both $col1$ and $col2$ are initialized to the first column in the base matrix. The first XOR matrix which is formed is shown in Figure 24(a). Similar to the case for T-neighborhood, the entry in the XOR-matrix $M(col1, col2)$ is obtained from the $col1$-th and $col2$-th columns of the base matrix. Step 1 is repeated again in Step 2, but all the entries in the XOR-matrices are complemented. The resulting background matrix is shown in Figure 25.

Proposition 8: Given an $(n,4)$-exhaustive code, construction 2 produces a code of length $2(1 + \frac{\lceil \log_2(\sqrt{n}) \rceil}{2} + \frac{\lceil \log_2(\sqrt{n}) \rceil^2}{2})^2$.

Proof: The outer for loop goes from $col1 = 1$ to $col1 = 1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$.

Similarly, the inner loop goes from $col2 = 1$ to $col2 = 1 + \lceil \log_2 \sqrt{n} \rceil +$

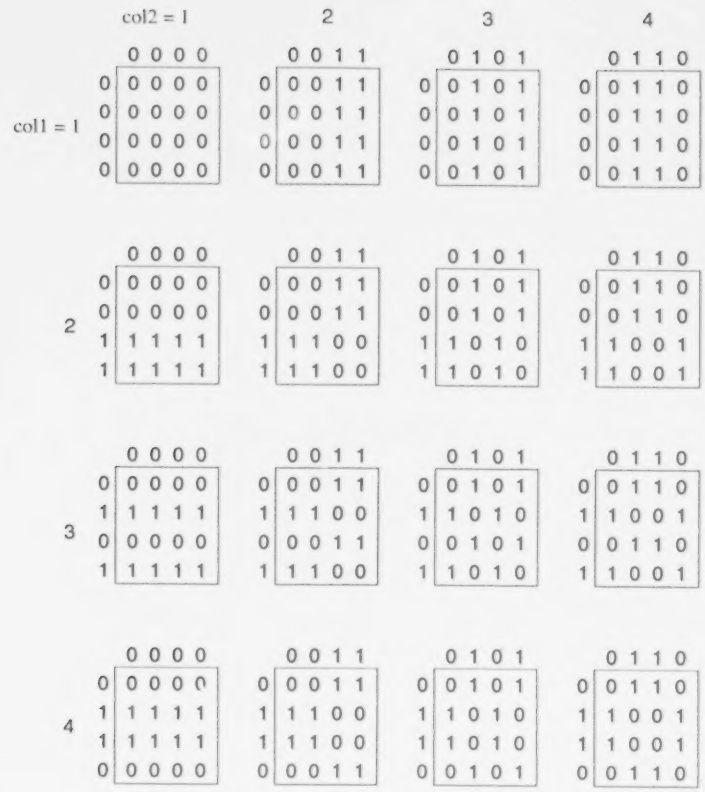


Figure 24: Construction of a $(16, 4)$-exhaustive Code With Respect to Scrambled Neighborhoods of Type 1 (Step 1)

$\frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2}$. Therefore the number L_1 of matrices generated in Step 1 is:

$$L_1 = (1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2})^2$$

The number of matrices generated in Step 2 is also L_1. Therefore, the length L of the code is:

$$L = 2(1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil (\lceil \log_2 \sqrt{n} \rceil - 1)}{2})^2$$

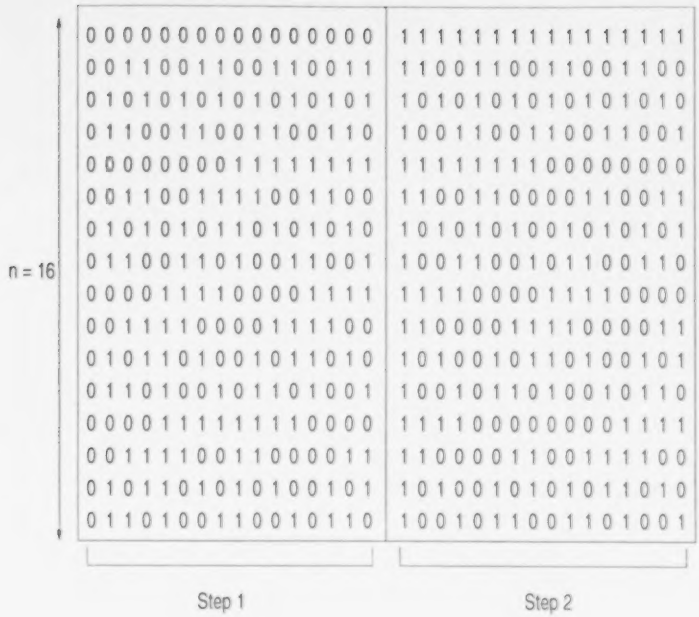


Figure 25: $(16, 4)$-exhaustive Code With Respect to Scrambled Neighborhoods of Type 1

$$= 2(1 + \lceil \log_2 \sqrt{n} \rceil + \frac{\lceil \log_2 \sqrt{n} \rceil^2}{2} - \frac{\lceil \log_2 \sqrt{n} \rceil}{2})^2$$
$$= 2(1 + \frac{\lceil \log_2 (\sqrt{n}) \rceil}{2} + \frac{\lceil \log_2 (\sqrt{n}) \rceil^2}{2})^2 \quad \square$$

Proposition 9: Construction 2 produces $(n, 4)$-exhaustive codes with respect to all scrambled neighborhoods of Type 1.

Proof: Consider an arbitrary scrambled neighborhood of type 1. Let the W, B and E cells be in columns a, b and c, respectively, where a, b and c are three distinct values in the range $0, \ldots, \sqrt{n} - 1$. We assume that the cell array is a $\sqrt{n} \times \sqrt{n}$ grid. The N and S cells are also in column b since they must be in the same column as cell B. Also, let the N, B and S cells be in rows d, e and f, respectively, where d, e and f are three distinct values in the range

$0, \ldots, \sqrt{n} - 1$. Consider two arbitrary backgrounds i and j where i and j

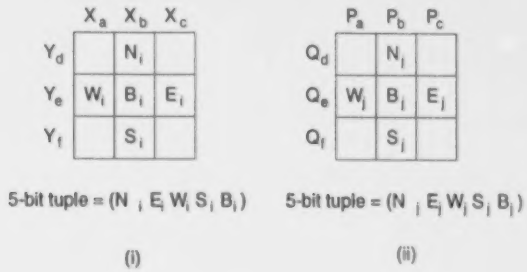


Figure 26: 5-cell Scrambled Physical Neighborhood

are two distinct values in the range $0, \ldots, m - 1$. The 5-cell neighborhoods after applying backgrounds i and j to the memory are shown in Figure 26. (In this figure we have again shown the neighborhood without scrambling for simplicity. The following argument is independent of the scrambling that may be present.)

Using Proposition 4, a base matrix which is the first half of a $(\sqrt{n}, 3)$-exhaustive code has at least four distinct 3-bit tuples and none of these tuples is the complement of the other three tuples. In Step 1 of code construction 2, the product of two vectors $col1$ and $col2$ is performed with $col1$ in the vertical direction and $col2$ in the vertical direction. Therefore, the 3-bit vector $col2(a)col2(b)col2(c)$ will go through at least four distinct states. Similarly, the $col1(d)col1(e)col1(f)$ vector will go through at least four distinct states.

Let S_1 be the set that contains the four distinct states for $col1(d)col1(e)col1(f)$. Let S_2 be the set that contains the four distinct states for $col2(a)col2(b)col2(c)$. Let $X = X_a X_b X_c$ be the state for the 3-bit vector $col2(a)col2(b)col2(c)$ for background i and $X \in S_2$. Let $Y = Y_d Y_e Y_f$ be the state for the 3-bit vector $col1(d)col1(e)col1(f)$ for background i and $Y \in S_1$. Similarly, let $P = P_a P_b P_c$ be the value for the 3-bit vector $col2(a)col2(b)c0l2(c)$ for back-

ground j and $P \in S_2$. Lastly, let $Q = Q_d Q_e Q_f$ be the value for the 3-bit tuple $col1(d)col1(e)col1(f)$ for background j and $Q \in S_1$.

Let i identify the background which must exist such that Y is a distinct tuple in S_1. Similarly, let j identify the background which must exist such that Q is a distinct tuple in S_1 and $Q \neq Y$. Then it must be true that $Y \neq \overline{Q}$ according to Proposition 4. We will prove that the 5-cell neighborhood N_i, E_i, W_i, S_i and B_i, obtained from the product of the two column vectors $X \times Y$, is distinct from the 5-cell neighborhood N_j, E_j, W_j, S_j and B_j, which is obtained from $P \times Q$. Furthermore, we will show that the 5-bit tuple $N_i E_i W_i S_i B_i \neq \overline{N_j E_j W_j S_j B_j}$.

Assume for a moment that $N_i E_i W_i S_i B_i = N_j E_j W_j S_j B_j$. Then,

$$X_b \oplus Y_d = P_b \oplus Q_d \tag{9}$$

$$X_b \oplus Y_e = P_b \oplus Q_e \tag{10}$$

$$X_b \oplus Y_f = P_b \oplus Q_f \tag{11}$$

$$X_a \oplus Y_e = P_a \oplus Q_e \tag{12}$$

$$X_c \oplus Y_e = P_c \oplus Q_e \tag{13}$$

Assume that $X_b \neq P_b$ implying that $P_b = \overline{X_b}$. From (9), (10) and (11), $Q_d = \overline{Y_d}$, $Q_e = \overline{Y_e}$ and $Q_f = \overline{Y_f}$ which is a contradiction since $Y_d Y_e Y_f \neq \overline{Q_d Q_e Q_f}$. Therefore, $X_b = P_b$. Using this result and equations (9), (10) and (11), $Y_d = Q_d$, $Y_e = Q_e$ and $Y_f = Q_f$. However, this is a contradiction since $Y_d Y_e Y_f \neq Q_d Q_e Q_f$. Therefore, we conclude that $N_i E_i W_i S_i B_i \neq N_j E_j W_j S_j B_j$.

Next we will show that $N_i E_i W_i S_i B_i \neq \overline{N_j E_j W_j S_j B_j}$. Assume for a moment that $N_i E_i W_i S_i B_i = \overline{N_j E_j W_j S_j B_j}$. Then,

$$X_b \oplus Y_d = \overline{P_b \oplus Q_d} \tag{14}$$

$$X_b \oplus Y_e = \overline{P_b \oplus Q_e} \quad (15)$$

$$X_b \oplus Y_f = \overline{P_b \oplus Q_f} \quad (16)$$

$$X_a \oplus Y_e = \overline{P_a \oplus Q_e} \quad (17)$$

$$X_c \oplus Y_e = \overline{P_c \oplus Q_e} \quad (18)$$

Assume that $X_b \neq P_b$ implying that $P_b = \overline{X_b}$. From (14), (15) and (16), $Q_d = Y_d$, $Q_e = Y_e$ and $Q_f = Y_f$ which is a contradiction since $Y_d Y_e Y_f \neq Q_d Q_e Q_f$. Therefore, $X_b = P_b$. Using this result and equations (14), (15) and (16), $Y_d = \overline{Q_d}$, $Y_e = \overline{Q_e}$ and $Y_f = \overline{Q_f}$. However, this is a contradiction since $Y_d Y_e Y_f \neq \overline{Q_d Q_e Q_f}$. Therefore, we conclude that $N_i E_i W_i S_i B_i \neq \overline{N_j E_j W_j S_j B_j}$. Using similar arguments, it can be shown that the 5-bit tuple obtained from $X \times Y$ is also distinct from the tuple obtained from $P \times Q$ when $X \neq P$ and $X \neq \overline{P}$.

We have proved that the 5-bit tuple obtained from the product of two vectors, $X \times Y$, is distinct from the 5-bit tuple obtained from the product of two vectors, $P \times Q$, where at least one member in the product is distinct from the corresponding member in the second product, i.e. either $Y \neq Q$ and $Y \neq \overline{Q}$, or $X \neq P$ and $X \neq \overline{P}$. We have also shown that the 3-bit vectors $col2(a)col2(b)col2(c)$ and $col1(d)col1(e)col1(f)$ will independently go through at least four distinct states. Therefore, in Step 1 of Construction 2, any 5-cell scrambled neighborhoods of type I will go through $4 \times 4 = 16$ distinct states. Since we have shown that $N_i E_i W_i S_i B_i \neq \overline{N_j E_j W_j S_j B_j}$ when either $Y \neq Q$ and $Y \neq \overline{Q}$, or $X \neq P$ and $X \neq \overline{P}$, by taking the complement of all the entries in XOR-matrices found in Step 1, the 5-cell scrambled neighborhoods of type 1 will go through the remaining 16 states. Taking the complement of the backgrounds from Step 1 is precisely what occurs in Step 2. Therefore any scrambled 5-cells neighborhood will go through 2^5 states, which proves that the code is $(n, 4)$-exhaustive with respect to scrambled neighborhoods

of Type 1. □

3.3 Transparent Near-Deterministic Tests for Detecting Single V-coupling Faults

In this M.Sc. project, the deterministic tests described in Section 3.1 were transformed into transparent near-deterministic tests using the technique described by Nicolaidis in [12]. This transformation consists of five steps which essentially ensure that each memory cell is complemented an even number of times. In a fault-free RAM, this procedure guarantees that the contents of the RAM after test application will be the same as the initial contents. Using the example in Figure 13, we will show how this 106-operation long deterministic test is transformed into a transparent test. The deterministic test can be divided into four kinds of subsegments. The first type is the $\Uparrow (w_0)$ initialization sequence. The second type is a $\Uparrow (r_b w_b r_b w_b)$ march. The third type is the background change $\Uparrow (r_b w_{\bar{b}})$. The second and third types are repeated until all the backgrounds in the background matrix have been applied to the RAM under test. The fourth type is the $\Uparrow (r_b)$ march at the end of the test. The four subsegment types will be labeled Init, S1, S2, and S3.

Nicolaidis' transformation from a regular deterministic test to a transparent near-deterministic test has four steps as follows:

Step 0: If the first operation in a subsegment type is a write operation, then add a read operation at the beginning of the subsegment type.

The first and only operation in the Init sequence is a w_0. Therefore, a read operation is inserted before the w_0 operation. The S1, S2 and S3 sequences each start with read operations and therefore do not need to be transformed. The in-

termediate algorithm $AL0$ obtained after applying Step 0 is shown in Figure 27.

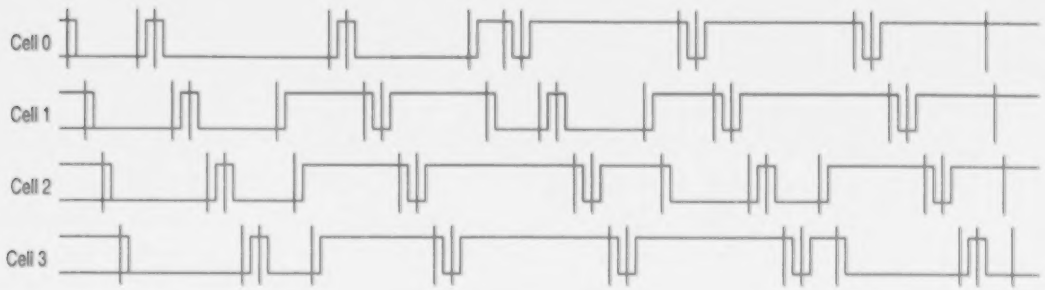


Figure 27: Test Structure After Applying Step 0: ALO

Step 1: If the initial algorithm includes an initialization sequence and if this sequence is useless for fault excitation, then delete this sequence from the algorithm ALO.

In our example, the initialization sequence is not required for fault excitation because this sequence is used only to initialize the RAM into a known state. Single V-coupling faults are triggered because the $(n, V-1)$-exhaustive background matrix guarantees that the $V-1$ cells involved in the fault will undergo all possible 2^{V-1} states. By removing the initialization sequence, we do not remove any of the possible 2^{V-1} states. Therefore the sequence can be deleted from $AL0$. The resulting algorithm $AL1$, is shown in Figure 28. In this example, we will assume that cells 0, 1, 2 and 3 initially contain 1, 1, 0, and 0, respectively.

The next step is to transform the march sequences S1 and background changes S2.

Step 2: Let a be the expected data of the first read operation to a cell i in $AL1$ and let b be the expected data of the first read operation to cell i in $AL0$. If $a = \bar{b}$, then change the data of all write operations in $AL1$ to be equal to

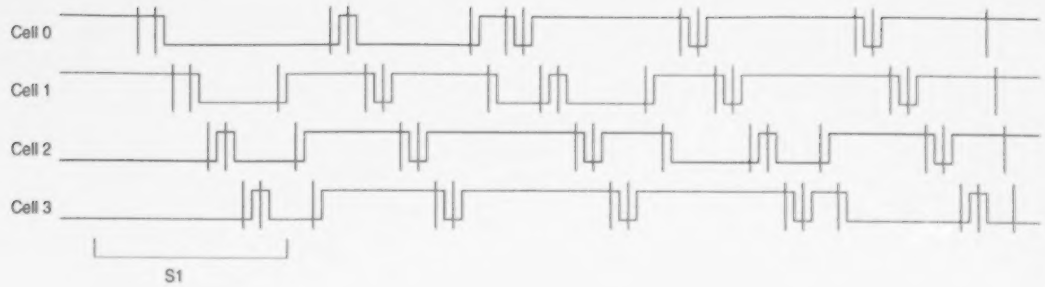


Figure 28: Test Structure After Applying Step 1, $AL1$

the complement of the data of the corresponding write operations in $AL0$. Similarly, change the expected data of the all the read operations (except the first read) in $AL1$ to be equal to the complement of the expected data of the corresponding read operations in $AL0$.

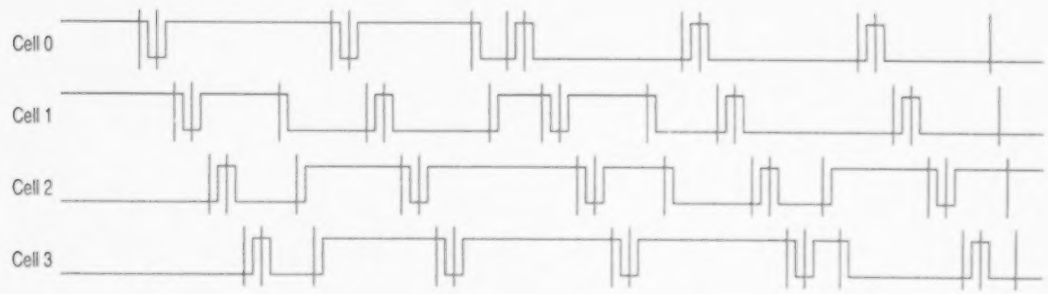


Figure 29: Test Structure After Applying Step 2, $AL2$

Applying Step 2 to the $AL1$ example in Figure 28, we obtain $AL2$ shown in Figure 29. We will use cell 0 to further explain this transformation step. In $AL1$, the first read operation in March 1 is r_1. This means that $a = 1$. In $AL0$, the first read operation is r_0 and therefore, $b = 0$. Since $a = \bar{b}$ we transformed the data of all of the read and write operations addressed to cell 0 in AL1, to the complement of the data of the corresponding read and write operations to cell 0 in AL0. Essentially, the waveform for cell 0 starting at the first write operation has

been flipped upside down. Cell 1 undergoes similar transformation. The waveforms for cells 2 and 3 are not transformed because the data of the first read for both cells in AL1 are equal to 0, which is the same as the data of the first read to the cells in AL0.

Step 3: If the last value written into a cell is equal to the complement of its initial value, then include a read and write complement pair of operations to that cell in a final background change sequence. This step will restore the contents of the RAM under test.

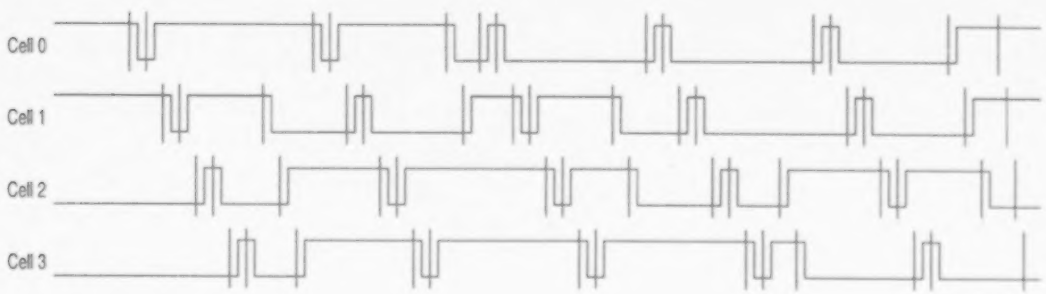


Figure 30: Test Structure After Applying Step 3, $AL3$

Figure 30 shows the algorithm $AL3$ obtained after applying Step 3 to $AL2$ from Figure 29. Note that a $r_b w_{\bar{b}}$ sequence has been added to cell 0 at the end of March 5. Similar operations are performed for cell 1 and cell 2.

Step 3 is equivalent to adding a new column to the end of the corresponding background matrix. The new augmented matrix is shown in Figure 31. The additional column has to be the same as the first column in the matrix. This is because the entries in the first column actually denote the initial contents of the memory. Adding the final column ensures that the final background change will restore the content of the RAM. No $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march need be associated with

this last background because this march operation will not trigger any new faults not already triggered in the first background.

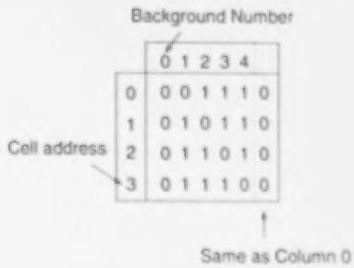


Figure 31: Background Matrix for a Transparent Test

Step 4: To obtain a signature, prefix a signature prediction phase to AL3. In this phase, perform only the read operations extracted from AL3.

Applying this step to the example in Figure 30 results in the transparent test shown in Figure 32.

Figure 32: Near-deterministic Transparent Test based on the Background Matrix in Figure 31

In the first phase, only the read operations are performed. Since the content of the memory is not changed in this phase, some of the values that are read may have to be complemented before being injected into the signature analyzer so that signatures calculated in phases 1 and 2 will be the same. The data obtained by a read has to be complemented if the corresponding read data in the second phase

is the inverse of the initial contents of the RAM. The signature obtained in the first phase is assumed to be the fault-free signature (a consequence of the fault-free read assumption). In the second phase, both the read and the write operations are performed. The read values are again injected into a signature analyzer to compute a second signature. This second signature is then compared with the fault-free signature to determine if the RAM is faulty: if the signatures are equal then the RAM is considered to be fault-free.

The fault coverage for the near-deterministic test is the same as the fault coverage for the deterministic test. This is because Nicolaidis' transformation technique does not reduce the fault coverage if the fault model used in the original test has the following symmetry property [12]:

Symmetry Property: Consider a fault model F and consider an arbitrary fault $f \in F$ which involves the states $s_1, s_2, \ldots, s_{k1}$ of some cells $C_1, C_2, \ldots, C_{k1}$ and the transitions $t_{k1+1}, t_{k1+2}, \ldots, t_{k1+k2}$ of some cells $C_{k1+1}, C_{k1+2}, \ldots, C_{k1+k2}$. Fault model F is symmetric if, for all $f \in F$, the faults which are derived from f by inverting any collection of the states $s_1, s_2, \ldots, s_{k1}$ and any collection of the transitions $t_{k1+1}, t_{k1+2}, \ldots, t_{k1+k2}$ of the above cells, also belong to F.

This symmetry property basically states that if a symmetrical fault model includes a fault that results a cell being forced to a faulty state b, then the fault model must include a similar fault that forces the same cell to a faulty state $\bar{b}$. Also, if a fault involving an $\uparrow$ transition in a cell is included in a fault model, then the same model must include a similar fault which involves a $\downarrow$ transition in the cell. For instance, if a fault model includes the 2-coupling fault $\uparrow i \Rightarrow \downarrow j$ but does not include $\uparrow i \Rightarrow \uparrow j$, then the symmetry property does not hold for that fault model. Similarly, if $\uparrow i \Rightarrow \downarrow j$ is in a symmetrical fault model, then the model must also

include a $\downarrow i \Rightarrow \downarrow j$. The single V-coupling fault model used in the deterministic test has this property because we consider both $\uparrow$ and $\downarrow$ transitions in the aggressor cell and also both state 0 and state 1 in the victim cell and surrounding neighborhood cells. In fact, most of the common fault models used in RAM testing satisfy the symmetry property.

3.4 An Improved Transparent Test for Detecting Single and Multiple V-coupling Faults

A transparent test constructed using Nicolaidis' transformation is vulnerable to the typically small but finite possibility of *aliasing*. This is the situation where a faulty RAM produces a response sequence that is different from that of a fault-free RAM, but the resulting compacted signatures are identical. Therefore, even though the fault coverage is not reduced by this transformation prior to response compression, a fault in the RAM may escape detection if aliasing occurs. The probability of aliasing can be greatly reduced if, instead of comparing two signatures once, different signature pairs are computed and compared many times when the test is applied. In this section we describe an improved transparent test that exploits this effect.

We will use a technique described in [10] to reduce the probability of aliasing. In this technique, a signature analyzer can be designed to be aliasing-free if the number of erroneous entries injected into the signature analyzer is bounded to some relatively small value. For instance, an LFSR with primitive characteristic polynomial of degree $[\log_2(n+1)]+2$ will have zero probability of aliasing if at most two erroneous entries are injected into the LFSR. Note that n here is the number of bits in the RAM under test. Therefore, to achieve guaranteed detection of single V-coupling faults, the transparent BIST test described in the previous

section has to be modified so that a signature comparison is performed after an interval that could possibly introduce no more than two errors.

The next step is to find out how many errors can occur in a march operation and in a background change operation. We will consider the march operation first. In the worst case, two errors can be injected into the signature analyzer. This is the case where a cell at a lower address triggers a fault in a cell at a higher address during a march operation. As an example consider two cells i and j where cell $i < j$.

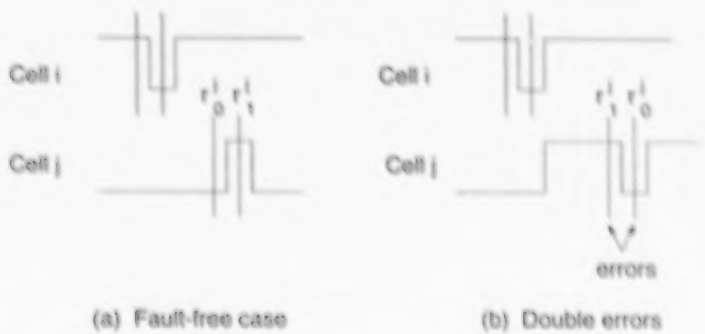


Figure 33: Maximum Number of Errors Observed During a March Sequence

Figure 33(a) shows the march sequence for cell i and cell j in the fault-free case. The two read operations for cell j are r_0^j followed by a r_1^j. Figure 33(b) shows the effect of a coupling fault where a 0 to 1 transition in cell i which causes cell j to change erroneously from 0 to 1. The two reads for cell j are r_1^j followed by a r_0^j. In the presence of the fault, both values read from cell j are erroneous.

Alternatively, we can find the maximum number of errors that can be caused by a single fault by counting the number of read operations that are performed to each cell in the RAM. If a single fault affects a memory cell, it can cause at most two errors during a march sequence because the cell is read twice during that march. Using the same arguments, at most one error is introduced during

a background change because each cell is read once. Using the above results, a signature comparison has to be performed at the end of a march and another comparison is required at the end of a background change. A single signature comparison cannot be used for a march and a background change because this would leave open the possibility of three errors and thus aliasing.

The improved transparent test is first of all divided into segments, where each segment corresponds to a background in the background matrix. Each segment has two phases: phase 1 and phase 2. Phase 1 is the signature generating phase where only the read operations extracted from phase 2 are performed. Phase 2 is the actual test phase where both the read and write operations based on a background are performed. In phase 1, the signature analyzer (*SA*) is first of all cleared. Next, the read operations are applied and the corresponding data values are injected into the SA. After the read operations extracted from the corresponding $(r_k w_{\bar{k}} r_{\bar{k}} w_k)$ march in phase 2 have been applied, the resulting content of the SA is stored as Signature 1. The SA then continues on and reads in the data from the remaining reads extracted from the background change. The content of the LFSR at the end of this process is stored as Signature 2.

At the start of phase 2, the SA is again cleared to the all-zero state. The march sequence is then applied to the RAM. Upon the completion of the march, the current state of the SA is compared with Signature 1. Next, the background change is performed. At the end of this operation, the SA is compared with Signature 2. When this modification is applied to the transparent test in Figure 32, we obtain the improved transparent test shown in Figure 34.

Some faults may still escape detection by this improved test. For example, if a cell at a higher address j triggers a fault a cell at a lower address i during a background change, it will not be detected since cell i will not be read again in that segment, as shown in Figure 35.

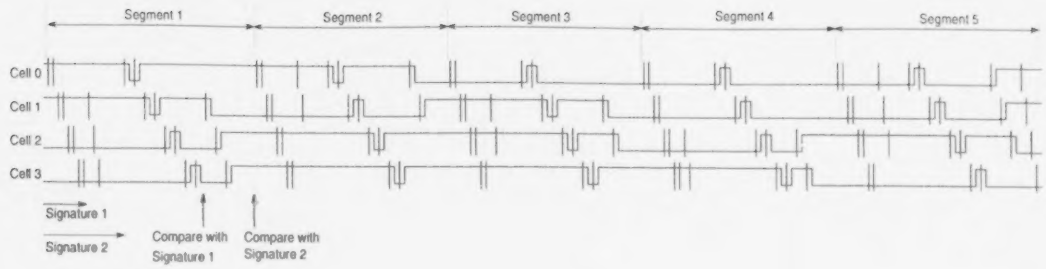


Figure 34: Transparent Test with Multiple Signature Generations and Comparisons

In this example both cell i and cell j are written by the background change. Figure 35(b) shows that cell i is not read again in the current segment after the fault has occurred. This means that the two signatures at the end of this segment are identical and the fault cannot be detected in this segment. Since the signature analyzer is cleared at the beginning of the next segment, this error information is lost and the fault may escape detection. To overcome this problem, we insert a read march at the end of each segment. It can be seen from Figures 35(c) and (d) that the additional read operation will detect the fault. This addition will increase the number of read operations during the background change and the final $\Uparrow$ (r_P) march to a maximum of two. This means there can be at most two errors during this period and therefore the signature analyzer will still be aliasing-free for single faults. The final aliasing-free transparent test obtained by inserting a read march at the end of each segment of the test in Figure 34 is shown in Figure 36. It contains 196 operations. Recall that the original test in Figure 13 contains 106 operations.

The pseudo-code for the aliasing-free transparent test is given in Figure 37. Note that the dimension of the background matrix BGM is $n \times (m + 1)$ even though there are only m backgrounds because of the addition of a final column

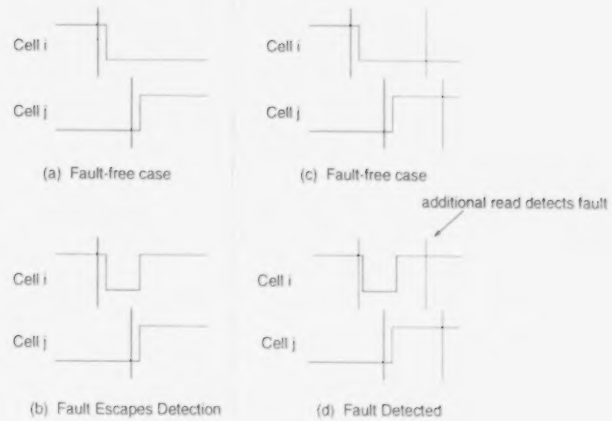


Figure 35: Example of a Fault that is Excited by a Background Change: (a) Fault-free Case; (b) Fault that Escapes Detection; (c) Fault-free Case with Additional Read Operations; (d) Fault Detected by Additional Read

that is used to restore the contents of the memory. We use c to denote the data returned by the first read operation in each segment.

Theorem 2: The test shown in Figure 36 is transparent and detects all single V-coupling faults if matrix BGM is an $(n, V-1)$-exhaustive code and the first and last columns of BGM are identical.

Proof: We first show that the test is transparent. Cells can be written at only

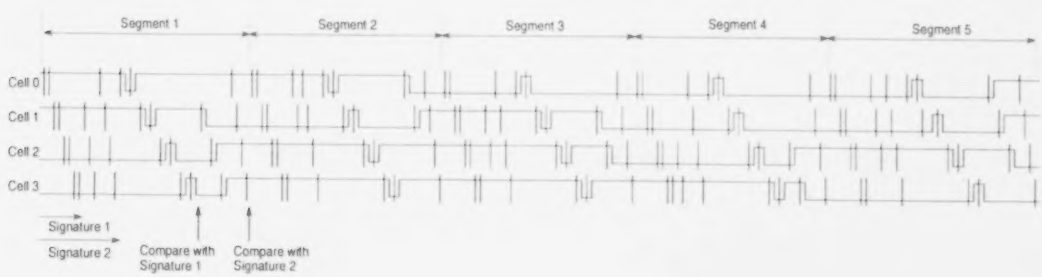


Figure 36: Aliasing-free Transparent Test

```
const n; /* number of RAM cells */
const m; /* number of backgrounds in background matrix */
const BGM[n,m + 1]; /* n × (m + 1) background matrix */
type address = 0 . . . (n-1); bit = (0,1), bg_position = 0, . . . m;
var a: address; bg_num: bg_position; P, N, c: bit; last_dbgr, set_fail: boolean;
var SA: LFSR; R1, R2: reg;
begin
    bg_num := 0;
    repeat
        /* Phase 1 */
        SA = 0; /* clear signature analyzer */
        for a := 0 to n − 1 do
            r^a_c; r^a_c; /* reads extracted from r_c w_c r_c w_c march */
        endfor;
        R1 = SA; /* store signature 1 */
        for a := 0 to n − 1 do /*reads from background change */
            P := BGM[a,bg_num]; N := BGM[a,bg_num + 1];
            if P ≠ N then r^a_c; endif;
        endfor;
        for a := 0 to n − 1 do /*read all cells at the end of a segment */
            P := BGM[a,bg_num]; N := BGM[a,bg_num + 1];
            r^a_{c⊕P⊕N};
        endfor;
        R2 = SA; /* store signature 2 */
        SA = 0; /* clear signature analyzer */
        /* Phase 2 */
        for a := 0 to n − 1 do
/* L1 */        r^a_c w^a_c r^a_c w^a_c; /* ascending r_c w_c r_c w_c march */
        endfor;
        if (SA ≠ R1) then set_fail := true; endif;
        for a := 0 to n − 1 do /* background change */
            P := BGM[a,bg_num]; N := BGM[a,bg_num + 1];
/* L2 */        if P ≠ N then r^a_c; w^a_c; endif;
        endfor;
        for a := 0 to n − 1 do /* read all cells at the end if a segment */
            P := BGM[a,bg_num]; N := BGM[a,bg_num + 1];
            r^a_{c⊕P⊕N};
        endfor;
        if (SA ≠ R2) then set_fail := true; endif;
        if bg_num = m − 1 then /* have reached last background */
            last_dbgr := true;
        else
            bg_num := bg_num + 1;
        endif;
    until last_dbgr;
end;
```

Figure 37: Aliasing-free Transparent Test Algorithm

lines $L1$ and $L2$ of the test algorithm (see Figure 37). The two writes from line $L1$ do not leave a cell complemented. The write from line $L2$ does complement the content of a cell if $P \neq N$. But, because the first and last columns of the BGM are identical, the situation $P \neq N$ will be encountered an even number of times for each row of BGM (and for each cell of the RAM). Therefore applying the test to a RAM complements the content of each cell an even number of times, and so the test is transparent.

At most two errors are injected into the SA during a march sequence because two read operations are performed during the march. From the result in [10], the signature obtained during this march sequence cannot be aliased. At most one error is injected into the SA during a background change because a maximum of one read is performed on a memory cell during the background change. At most one error is injected into the SA during the final $\Uparrow (r_P)$ march at the end of a segment because each cell is read once during this march. From [10], the signature computed during a background change and the final $\Uparrow (r_P)$ march cannot be aliased [10]. Therefore, the test is aliasing-free for single V-coupling faults. From Theorem 1, the deterministic test in Figure 13 detects all single V-coupling faults. The fault coverage is not reduced by Nicolaidis's transformation because the single V-coupling fault model is symmetrical. Since test is aliasing-free, the test in Figure 36 detects all single V-coupling faults. Therefore the test is transparent and detects all single V-coupling faults. □

Theorem 3: The test shown in Figure 36 has an expected length of $9.5mn$ where m is the number of backgrounds in the background matrix and n is the number of bits in the RAM.

Proof: Assume the entries in the background matrix are random. Consider the number of read operations addressed to each cell in phase 1 of a segment. In this phase, only the read operations extracted from phase 2 are performed. There are two reads extracted from the march sequence. Consider the number of reads extracted from the background change. If the P bit and N bit of a cell are different, then the cell is read once. If the P bit is equal to the N bit, then no read operations are applied to the cell. If the entries of the background matrix are random, then the probability that P bit $\neq$ N bit is $\frac{1}{2}$. Finally, all the cells are read again at the end of phase 1. Therefore, the number of operations per cell in phase 1 of a segment is equal to $2+\frac{1}{2}+1 = 3.5$ operations.

Consider the number of read and write operations performed on a cell in phase 2 of a segment. The $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march sequence has 4 operations. A cell is read once and written once if its P bit is not equal to its N bit. Finally, all cells are read again at the end of phase 2. Therefore, the number of operations in phase 2 of a segment is equal to $4+(\frac{1}{2}\times 2)+1 = 6$ operations.

The number of segments is equal to the number of backgrounds in a background matrix, m. Therefore, each cell undergoes $(3.5 + 6)m = 9.5m$ operations. There are n cells in the RAM, so the length of the test is $9.5mn$. □

Table 3 shows the length of the transparent test for $V = 2$, 3, and 4-coupling faults. $V = 5t$ refers to a PNPSF with respect to the scrambled T-neighborhoods while $V = 5x$ refers to a PNPSF with respect to scrambled neighborhoods of type 1. When $V = 2$ the test construction uses a trivial $(n, 1)$-exhaustive code where each codeword is 01.

V	Code Length, m	Proposition	Test Length Order
2	2	-	$O(n)$
3	$\lceil \log_2 n \rceil + \lceil \log_2(\lceil \log_2 n \rceil + 1) \rceil + 2$	1	$O(n \log_2 n)$
4	$\lceil \log_2 n \rceil^2 + \lceil \log_2 n \rceil + 2$	3	$O(n(\log_2 n)^2)$
$5t$	$2\lceil \log_2 \sqrt{n} \rceil^2 (1 + \lceil \log_2 \sqrt{n} \rceil)$	5	$O(n(\log_2 n)^3)$
$5x$	$2(\frac{\lceil \log_2(\sqrt{n}) \rceil^2}{2} + \frac{\lceil \log_2(\sqrt{n}) \rceil}{2} + 1)^2$	8	$O(n(\log_2 n)^4)$

Table 3: Transparent Test Length

3.5 Analysis of the Probability of Aliasing

In the previous subsection, we noted that the transparent BIST scheme detects all single V-coupling faults if an LFSR with a characteristic primitive polynomial of degree $\lceil \log_2(n+1) \rceil + 2$ is used as the signature analyzer. However, if there is more than one single $V-$coupling fault present in the RAM under test, then there is the possibility that these faults will not be detected. Faults can escape detection if they are triggered in the one or more of the same segments. Consider the case where two single $V-$coupling faults are present. If the same background that triggers a fault in a memory cell also triggers a second fault in another cell, the maximum number of errors that can occur during the march sequence is four, i.e. two errors for the first cell and two errors for the second cell (refer to Figure 33 which shows how the errors occur for one cell). The LFSR described above is aliasing-free for at most two errors at a time, therefore two faults might cause aliasing and thus escape detection. The probability that two faults will not be detected in the proposed transparent scheme is called the *escape probability.*

Consider two faults A and B. Let X be the set of backgrounds that detect fault A and Y be the set of backgrounds that detect fault B. Faults A and B may

escape detection if $X = Y$, i.e. if the same backgrounds that detect fault A are also the only backgrounds that detects fault B. If $X \neq Y$, then there exist one or more backgrounds that detect A but not B, or vice versa. This means that in one or more segments in the transparent test, one of the faults A or B will be triggered, but not the other. Therefore the number of errors in these segments is no more than two, which eliminates the possibility of aliasing in those segments; thus the fault will be detected. Detection of the second fault is not necessary because an error flag will be set when the first fault is detected, indicating that the RAM is faulty.

Theorem 4: The escape probability EP for two single $V-$coupling faults A and B is as follows:

$$EP = (2^{V-1}-1)^{2m}\, 2^{-2m(V-1)} \sum_{i=1}^{m-(2^{V-1}-1)} \left[\frac{1}{(2^{V-1}-1)^2}\right]^i \begin{pmatrix} m \\ i \end{pmatrix}$$

Proof: The entries of the background matrix are assumed to be random. Let T_A be the $(V-1)$-bit tuple that allows fault A to be detected and let T_B be the $(V-1)$-bit tuple that allows fault B to be detected. In order to establish the size of this probability we have to consider the number of backgrounds in the (n, V)-exhaustive code that can contain both T_A and T_B. Since the background matrix is (n, V)-exhaustive, then any combination of $V-1$ rows will have all 2^{V-1} binary $(V-1)$-tuples as column vectors. Since there are 2^{V-1} distinct tuples, any particular $(V-1)$-tuple can occur at most $m-(2^{V-1}-1)$ times. Therefore, T_A and T_B can each occur at most $m-(2^{V-1}-1)$ times. Faults A and B may escape detection if T_A and T_B only occur in the same backgrounds. Therefore, the escape probability EP is the sum of the probability that T_A is in exactly i backgrounds and the probability that T_B only occurs in the same i backgrounds, for $i = 1, 2, \ldots, m-(2^{V-1}-1)$.

The escape probability is thus given by the following sum:

EP = (Probability of only 1 background detecting fault A) $\times$ (probability that the same background is the only background that detects fault B)
+(Probability of only 2 backgrounds detecting fault A) $\times$ (probability that those 2 backgrounds are the only backgrounds that detect fault B)
+(Probability of only 3 backgrounds detecting fault A) $\times$ (probability that those 3 backgrounds are the only backgrounds that detect fault B)
+

.

.

+(Probability of only i backgrounds detecting fault A) $\times$ (probability that those i backgrounds are the only backgrounds that detect fault B)
+

.

.

+(Probability of only $m - [2^{V-1} - 1]$ backgrounds detecting fault A) $\times$ (probability that those $m - [2^{V-1} - 1]$ backgrounds are the only backgrounds that detect fault B)

Consider the i-th term in the above summation. There are two events to consider. The first event (Event 1) is "only i backgrounds detect fault A". The probability P_1 for Event 1 is given by:
P_1 = Probability that i backgrounds detect fault A $\times$ Probability that $m - i$ backgrounds do not detect fault A $\times \begin{pmatrix} m \\ i \end{pmatrix}$.

The $\begin{pmatrix} m \\ i \end{pmatrix}$ term is present because in this event, the i backgrounds can be any columns from the background matrix. Since there are m backgrounds in the

matrix, then there are $\begin{pmatrix} m \\ i \end{pmatrix}$ possible combinations of i backgrounds out of m.

The probability that i backgrounds detects fault A is equal to (Probability that a background detects fault $A)^i$. The probability P_A that a background will detect fault A is the probability that T_A occurs in a background. Since there are 2^{V-1} distinct $(V-1)$-bit tuples, then P_A is $\frac{1}{2^{V-1}}$.

The probability that $m-i$ backgrounds do not detect fault A is (Probability that a background does not detect fault $A)^{m-i}$, where the probability that a background does not detect fault A is 1- P_A $=1-\frac{1}{2^{V-1}}=\frac{2^{V-1}-1}{2^{V-1}}$

Therefore, $P_1=(\frac{1}{2^{V-1}})^i(\frac{2^{V-1}-1}{2^{V-1}})^{m-i}\begin{pmatrix} m \\ i \end{pmatrix}$.

The second event (Event 2) is "the same i backgrounds are the only backgrounds that detect fault B". The probability P_2 for Event 2 is similar to P_1 except that in this case, we omit the $\begin{pmatrix} m \\ i \end{pmatrix}$ term because in Event 1, the set of i backgrounds has been chosen. P_2 is given by:

P_2 = Probability that the i known backgrounds detect fault B $\times$ Probability that the $m-i$ remaining backgrounds do not detect fault B

$= (P_B)^i \times (1-P_B)^{m-i}$,

where P_B is the probability that a background detects fault B. P_B is equal to the probability that T_B occurs in a background and therefore, $P_B = P_A = \frac{1}{2^{V-1}-1}$ Substituting P_B into the equation for P_2 we obtain: $P_2=(\frac{1}{2^{V-1}})^i(\frac{2^{V-1}-1}{2^{V-1}})^{m-i}$.

Therefore the i-th term of EP can be expressed as:

$$EP_i=(\frac{1}{2^{V-1}})^i(\frac{2^{V-1}-1}{2^{V-1}})^{m-i}\begin{pmatrix} m \\ i \end{pmatrix}\times(\frac{1}{2^{V-1}})^i(\frac{2^{V-1}-1}{2^{V-1}})^{m-i}$$

Summing over all possible values of i, the escape probability EP is:

$$\begin{aligned} EP &= \sum_{i=1}^{m-(2^{V-1}-1)} \left(\frac{1}{2^{V-1}}\right)^{2i} \left(\frac{2^{V-1}-1}{2^{V-1}}\right)^{2m-2i} \binom{m}{i} \\ &= \left[\frac{2^{V-1}-1}{2^{V-1}}\right]^{2m} \sum_{i=1}^{m-(2^{V-1}-1)} \left[\frac{1}{(2^{V-1}-1)^2}\right]^{i} \binom{m}{i} \quad \square \end{aligned}$$

Proposition 10: An upper bound on EP is:

$$EP < \left[\frac{2^{V-1}-1}{2^{V-1}}\right]^{2m} \left[\left(1+\frac{1}{(2^{V-1}-1)^2}\right)^{m} - 1\right].$$

Proof: We can approximate EP using the Binomial Theorem. The Binomial theorem states that for each natural number c:

$$(1+z)^c = \sum_{i=0}^{c} \binom{c}{i} z^i$$

Thus

$$(1+z)^c - 1 = \sum_{i=1}^{c} \binom{c}{i} z^i$$

This theorem cannot be applied directly to EP because the upper limit of the summation, $m-(2^{V-1}-1)$ is different from the m in $\binom{m}{i}$. In order to use the closed form result, the upper limit of the summation is set to m instead of $m-2^{V-1}-1$. This approximation of EP, EP_{approx}, is slightly higher that the actual escape probability. Therefore,

$$\begin{aligned} EP &< \left[\frac{2^{V-1}-1}{2^{V-1}}\right]^{2m} \sum_{i=1}^{m} \left[\frac{1}{(2^{V-1}-1)^2}\right]^{i} \binom{m}{i} \\ &< \left[\frac{2^{V-1}-1}{2^{V-1}}\right]^{2m} \left[\left(1+\frac{1}{(2^{V-1}-1)^2}\right)^{m} - 1\right] \quad \square \end{aligned}$$

Example: The exact escape probability for $V = 3$ is:

$$\begin{aligned} EP(V=3) &= (\frac{3}{4})^{2m} \sum_{i=1}^{m-3} (\frac{1}{9})^i \begin{pmatrix} m \\ i \end{pmatrix} \\ &= (\frac{9}{16})^m \sum_{i=1}^{m-3} T(m,i) \end{aligned}$$

where $T(m,i) = (\frac{1}{9})^i \begin{pmatrix} m \\ i \end{pmatrix}$. Using the Binomial Theorem, $EP(V=3)$ can be written as:

$$\begin{aligned} EP(V=3) &= (\frac{9}{16})^m \left[(1+\frac{1}{9})^m - 1 - T(m,m) - T(m,m-1) - T(m,m-2)\right] \\ &= (\frac{9}{16})^m \left[(1+\frac{1}{9})^m - 1 - (\frac{1}{9})^m - m(\frac{1}{9})^{m-1} - \frac{m(m-1)}{2}(\frac{1}{9})^{m-2}\right] \\ &= (\frac{9}{16})^m \left[(\frac{10}{9})^m - 1 - (\frac{1}{9})^{m-2}(\frac{1}{81} + \frac{m}{9} + \frac{m(m-1)}{2})\right] \end{aligned}$$

The approximation of the escape probability for $V = 3$ (using the upper bound from Proposition 10) is given as:

$$\begin{aligned} EP_{approx}(V=3) &= (\frac{3}{4})^{2m} \left[\left(1+\frac{1}{3^2}\right)^m - 1\right] \\ &= (\frac{9}{16})^m \left[\left(\frac{10}{9}\right)^m - 1\right] \end{aligned}$$

For $V = 3$, the number m of backgrounds as a function of RAM size n is $\lceil \log_2 n \rceil + \lceil \log_2(\lceil \log_2 n \rceil + 1) \rceil + 2$ according to Proposition 1. Figure 38(i) shows EP and EP_{approx} for $V = 3$. The two probabilities agree with very small differences. The differences in EP and EP_{approx} for $V = 3$ is shown in Figure 38(ii).

From Figure 38(ii), the $EP_{approx}(V = 3)$ is higher than $EP(V = 3)$ by about 8.5×10^{-29} for RAM size $n = 1M$.

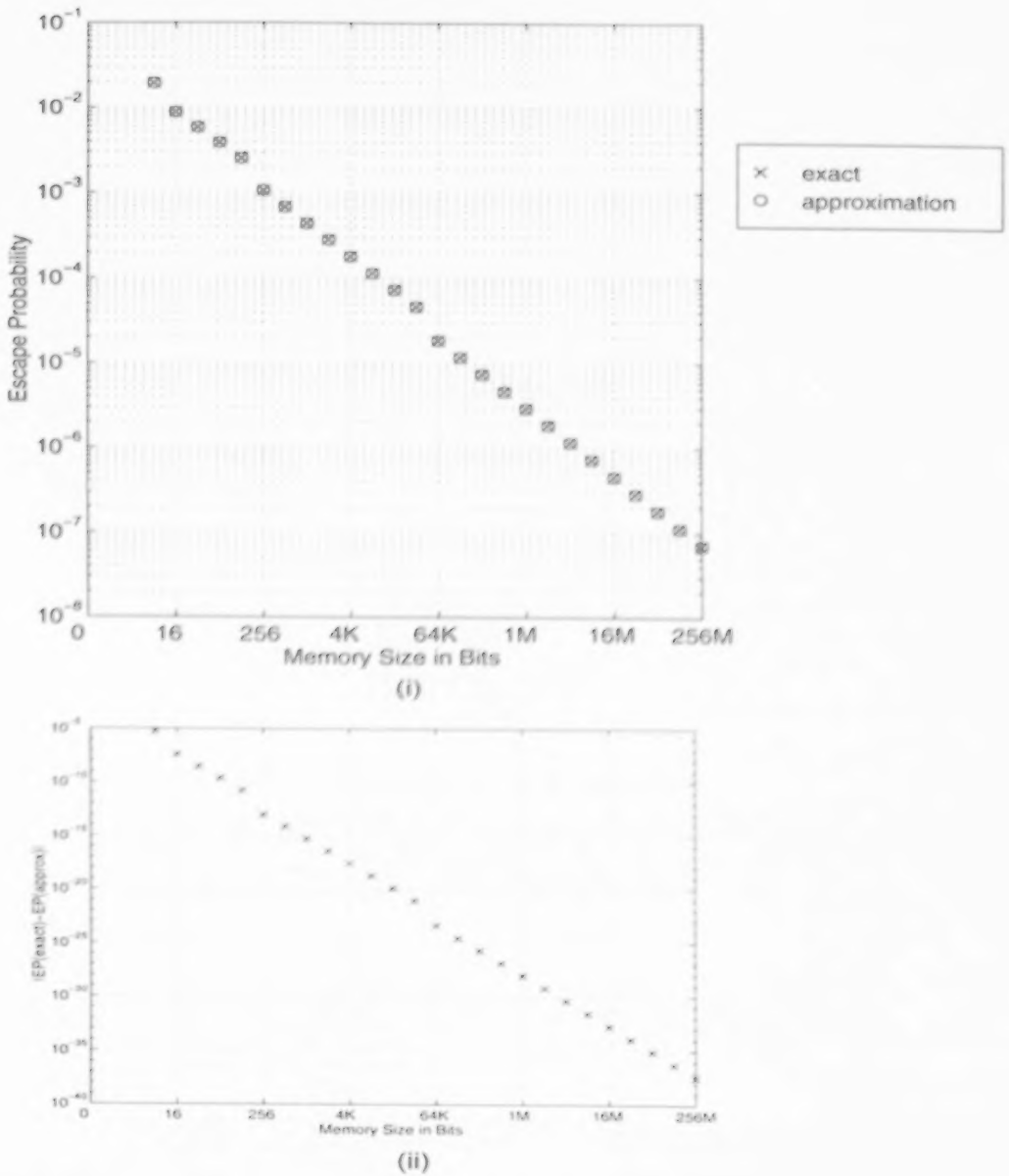


Figure 38: (i) The Exact and Approximate Escape Probability for Typical RAM Sizes for V=3; (ii) Difference Between the Exact and Approximate Escape Probability

Figure 39 shows the probability that the errors caused by two single V-coupling faults (V = 2, 3, 4, 5t and 5x) will escape detection because of aliasing (EP_{approx}) for typical RAM sizes. The escape probabilities are very small. For a $1M$ RAM,

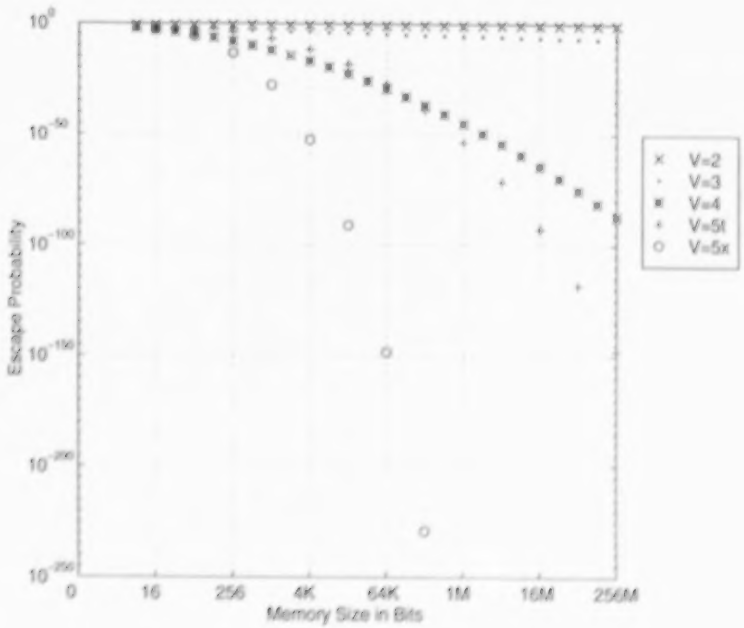


Figure 39: The Escape Probability for V = 2, 3, 4, $5t$ and $5x$

the escape probabilities are 0.1875, 3×10^{-6} and 6×10^{-46} for two single 2, 3 and 4-coupling faults, respectively. Note that the curves for $V = 5t$ and $V = 5x$ are different in Figure 39. This is because the codes for $V = 5x$ require many more backgrounds than the codes for $V = 5t$. The probability that two faults will not be detected is reduced even more by the finite probability of aliasing due to the LFSR. The signature analyzer used in the transparent BIST scheme is an LFSR with a characteristic primitive polynomial of degree $\lceil \log_2(n+1) \rceil + 2$. The probability of aliasing for this LFSR is approximately equal to $2^{-(\lceil \log_2(n+1) \rceil + 2)}$. Therefore, when

two single 3-coupling faults are present in a $1M$ RAM, the probability of aliasing for the transparent BIST RAM, $P_{(n=1M,V=3)}$ is approximately:

$$\begin{aligned} P_{(n=1M,V=3)} &= 2^{-(\lceil \log_2(1M+1) \rceil + 2)} \times 3 \times 10^{-6} \\ &= 1.4 \times 10^{-12} \end{aligned}$$

4 Detailed Design Description

In this chapter, the BIST RAM design is presented in detail. An earlier version of this design is described in [24]. The current design will be presented in [25]. The overall BIST RAM architecture, which consists of the $n \times 1$ RAM, the BIST controller and the Data Path, is described in Section 4.1. This section also discusses the BIST RAM operation and the interface between the BIST RAM and the external environment. The detailed descriptions of the BIST controller and the data path are given in Sections 4.2 and 4.3, respectively.

4.1 BIST RAM Architecture

The BIST RAM consists of three key architectural elements: the $n \times 1$ RAM, the BIST Controller and the Data Path. Figure 40 shows the block diagram of the BIST RAM.

The $n \times 1$ RAM has a read input, a write input, a $(\log_2 n)$-bit wide address bus input and a 1-bit-wide bi-directional data bus. The BIST controller is a state machine that controls the RAM and the data path when the self-test is applied. The data path has five major hardware blocks: address generator, background counter, background code logic, response analyzer and test pattern generator.

The BIST RAM has two operating modes: normal mode and self-test mode (A more appropriate name for the self-test mode is "BIST mode" since the BIST circuitry itself is not being tested). When the BIST RAM is in normal mode, the BIST circuitry is de-activated and the $n \times 1$ RAM performs normal read and write operations as requested by the external environment. In the self-test mode, the RAM is isolated from the external environment and the transparent test is applied to the RAM under the direction of the BIST controller.

The BIST RAM interfaces with the external environment via five signals:

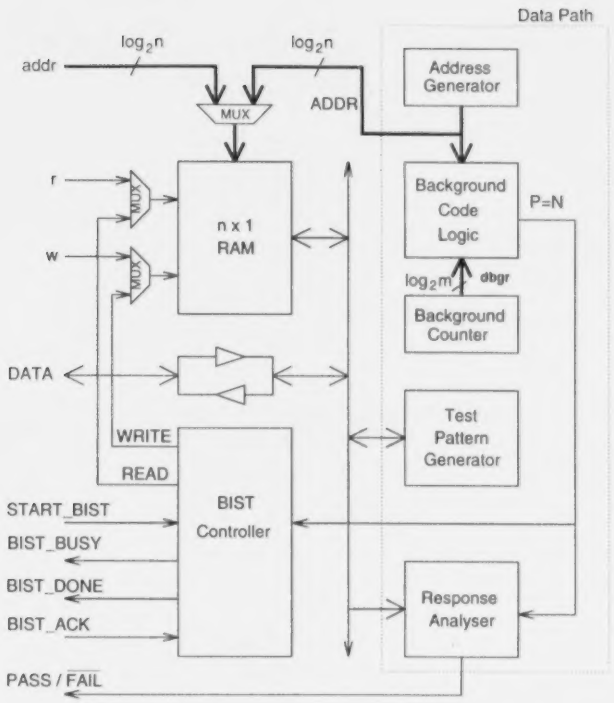


Figure 40: Simplified Block Diagram of BIST RAM

START_BIST, BIST_BUSY, BIST_DONE, BIST_ACK and PASS/$\overline{\text{FAIL}}$. When START_BIST signal is asserted high, the BIST RAM goes into self-test mode and performs the transparent test. The BIST_BUSY output is used to indicate that the self-test is executing. The BIST_DONE output is asserted high when the transparent test is completed. If no fault is detected during the test, the PASS/$\overline{\text{FAIL}}$ output will remain high; otherwise, this output will be set to low. The BIST_DONE signal will remain asserted high until the BIST RAM receives the BIST_ACK from the external environment. When the asserted BIST_ACK signal is received, the BIST_BUSY and BIST_DONE signals are de-asserted and the BIST RAM returns to normal mode.

Three multiplexers are used to isolate the RAM during self-test mode. The

BIST_BUSY signal is used to control the multiplexers since BIST_BUSY is set to 0 in the normal mode and is set to 1 in the self-test mode. Therefore when BIST_BUSY = 0 the multiplexers select the externally supplied inputs addr, r and w to be sent to the RAM. When BIST_BUSY = 1, the multiplexers select the ADDR, READ and WRITE signals generated internally by the BIST controller.

The VHDL descriptions and the schematics for the BIST RAM are included in Appendix A and Appendix B, respectively. The BIST controller and data path are described in detail in the next two subsections.

4.2 BIST Controller

In this project, a few alternate designs that implement the BIST scheme were considered before the final design was chosen. One of the alternatives that was studied is a design that enables both the P bits and the N bits to be generated using one common circuit. In our final design, we chose to use two separate circuits for generating the P and the N bits. We have thus traded-off the reduction in area overhead in the alternate design for shorter test times in the final design. This is because we would have to add one extra state in our BIST controller to execute the background change operations. The extra state is required because the common circuit can only generate one of the two bits at a time. Using two separate circuits for the P and N bits enables both bits to be generated in parallel, which leads to shorter test times in the final design.

The BIST controller implements the transparent test algorithm by sending control signals to the RAM and the hardware blocks in the data path. Figure 41 shows the flowchart for the BIST controller. The rectangular boxes represent controller states. Signals listed in a box are asserted during the corresponding state. The rounded boxes represent conditional states. These conditional states

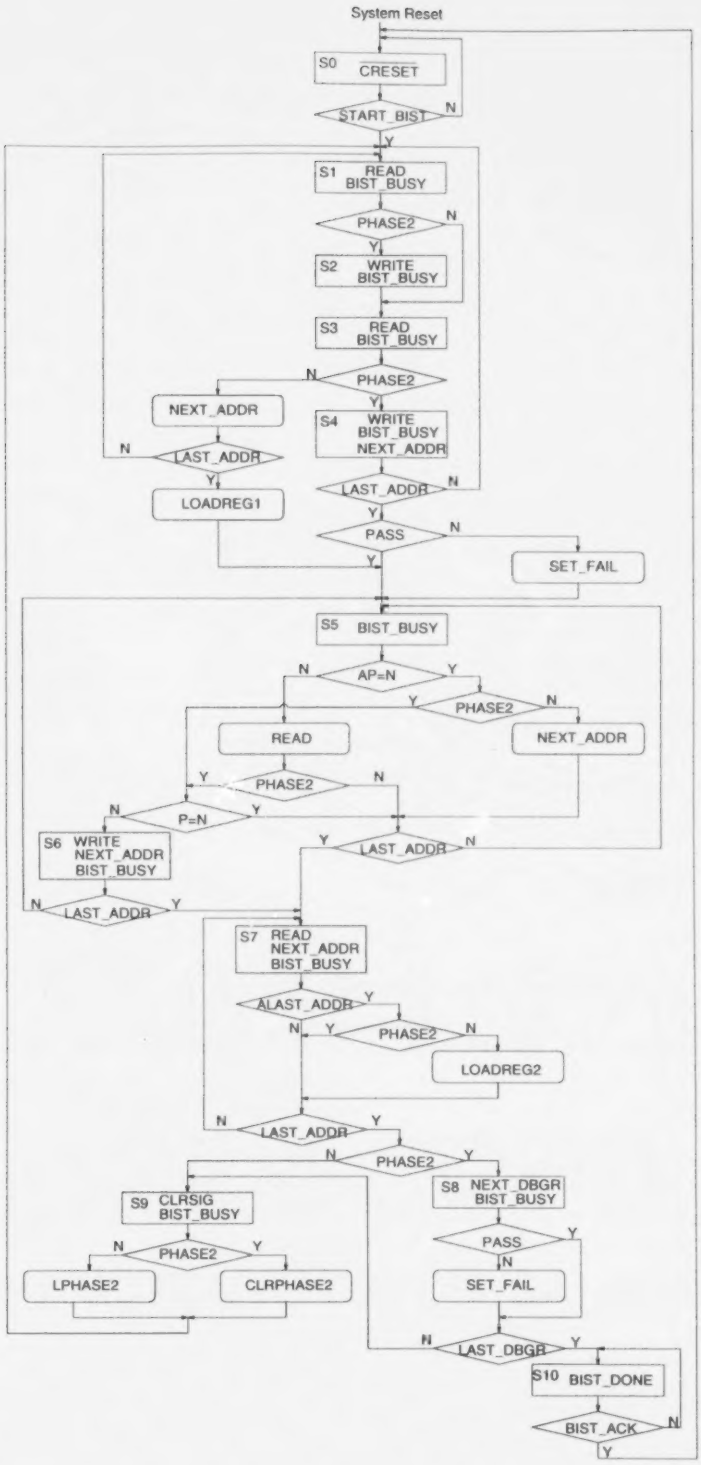


Figure 41: The BIST Controller Flow Chart

are removed in the final design where they are changed to conditional outputs. The diamonds give decision conditions for the next state transitions. Table 4 and Figure 42 list the control and status signals connected to the BIST controller.

The $\overline{\text{CRESET}}$ signal is the local reset output. (The overline indicates an active low signal.) This output is used to clear all the non-RAM flip-flops in the BIST RAM. In each segment of the transparent test, the PHASE2 latch is cleared in phase 1 and set in phase 2. In the flowchart, this flag is used to bypass all the write operations during phase 1. The LAST_ADDR signal from the data path is used to indicate if the last cell address has been reached during either the $\Uparrow (r_P w_P r_P w_P)$ march or a background change sequence. The LAST_DBGR signal is asserted if the last background has been loaded into the RAM. If the P bit for a cell is the same as its N bit, the P=N signal is asserted. This signal is used to determine if cells contents have to be changed during a a background change. The PASS signal from the data path indicates if the two signatures computed during phase 2 of a segment are the same as Signature 1 and Signature 2 generated in phase 1 of the same segment. If one or both of the signatures differ from the corresponding signatures obtained in phase 1, then the PASS signal is cleared by the response analyzer. The SET_FAIL output from the controller is then used to clear the PASS/$\overline{\text{FAIL}}$ latch, i.e. if PASS = 0, then SET_FAIL is asserted which in turns clear the PASS/$\overline{\text{FAIL}}$ latch to 0. This is done to ensure that the external PASS/$\overline{\text{FAIL}}$ output of the BIST RAM will stay cleared (i.e. FAIL active) throughout the remainder of the test even if no further faults are detected in subsequent segments. Note that PASS/$\overline{\text{FAIL}}$ is set to 1 initially by the controller reset output $\overline{\text{CRESET}}$. Therefore, if the two signatures computed in phase 2 are identical to Signature 1 and Signature 2 for all the test segments, the PASS/$\overline{\text{FAIL}}$ latch will remain set at the end of the test, indicating that the RAM has passed the test. Note also that the transparent test does not stop after the first fault is detected but continues on to the end. This

way only the contents of the faulty cells may be destroyed while the contents of the remaining cells are restored at the end of the test.

The BIST controller READ and WRITE outputs are asserted according to the corresponding read and write operations in the transparent test algorithm. The NEXT_ADDR signal is used to increment the address pointer for the ⇑⇑ marches and background changes. The NEXT_DBGR signal is used to increment the background number. LOADREG1 and LOADREG2 are control signals that are asserted during phase 1 of a test segment. When LOADREG1 is asserted, the content of the signature analyzer is stored in a register as Signature 1. Similarly, LOADREG2 is used to store Signature 2. The CLRSIG output is used to clear the signature analyzer at the end of phase 1 and phase 2. Note that the signature analyzer is not cleared after Signature 1 is stored. Two outputs from the controller are used to control the PHASE2 latch. LPHASE2 is asserted at the end of phase 1 to set PHASE2 to 1. CLRPHASE2 is asserted at the end of phase 2 to clear the latch.

The state diagram obtained from the flow chart is shown in Figure 43. The circles in the diagram represent states. The signals listed (in italics) beside a circle are asserted during the corresponding state. The signals are asserted as soon as the BIST controller enters the state and it is de-asserted when the controller exits that state. For example, the timing diagram for the WRITE signal asserted during state S2 is shown in Figure 44. The WRITE signal remains high for at least one clock period and the RAM can response to this signal during the period.

The arcs in the state diagram represent state transitions and the signals listed on the arcs are the conditions for the state transitions. The conditional states in the flow chart have been removed and the signals that are asserted in these states have been transformed into conditional outputs. For example, in state S3 output LOADREG1 is asserted if PHASE2 = 0 and LAST_ADDR = 1. In the

Input	Description
START_BIST	BIST enable signal from external environment
BIST_ACK	BIST acknowledge signal from external environment
PHASE2	equal to 0 in phase 1; equal to 1 in phase 2
LAST_ADDR	equal to 1 if last address in RAM has reached
LAST_DBGR	equal to 1 if last background has been applied to RAM
P=N	equal to 1 if the P and N bits are the same
PASS	equal to 1 if the signatures computed in phase 2 are the same the signatures computed in phase 1
Output	**Description**
READ	BIST controller read signal
WRITE	BIST controller write signal
BIST_BUSY	asserted when BIST routine is executing
BIST_DONE	asserted when BIST routine has completed
$\overline{\text{CRESET}}$	BIST controller reset
NEXT_ADDR	increment address signal
NEXT_DBGR	increment background number signal
LOADREG1	load register 1 signal
LOADREG2	load register 2 signal
CLRSIG	reset signature analyzer signal
LPHASE2	set PHASE2 latch signal
CLRPHASE2	clear PHASE2 latch signal
SET_FAIL	set PASS/$\overline{\text{FAIL}}$ latch
STATE	BIST controller current state

Table 4: BIST Controller Inputs and Outputs

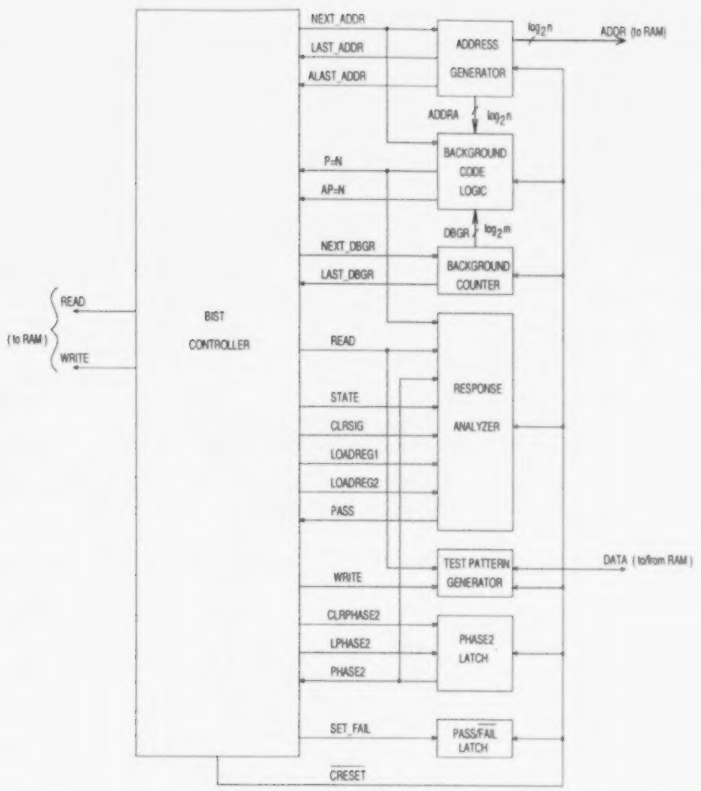


Figure 42: Interface Between the BIST Controller and the Data Path

state diagram design, both the BIST controller and the data path are synchronized to the same (falling) clock edge. This approach is used to increase the operating speed of the BIST scheme because both the controller and datapath operate in parallel. Since the controller and the data path are clocked with the same edge, the unlatched outputs from some of the hardware blocks in the data path are used immediately to decide next state transitions for some of the states in the state diagram. These "early" outputs are the AP=N and ALAST_ADDR signals, which are the unlatched variants of the P=N and LAST_ADDR.

There are eleven states in the BIST controller's state diagram, as shown in

Figure 43: The BIST Controller State Diagram

Figure 43. After power-on, the BIST controller is initialized to state S0. This is assumed to be done by a system reset signal. S0 corresponds to the normal mode of the BIST RAM. The BIST RAM behaves like a normal RAM in this state and it responds to the external addr, r, and w system inputs. In S0 the BIST controller reset output, $\overline{\text{CRESET}}$, is asserted. This signal is used to clear all the non-RAM flip-flops in the BIST RAM (except for the flip-flops for the state variables, which themselves are cleared by the system reset). The BIST controller will stay in

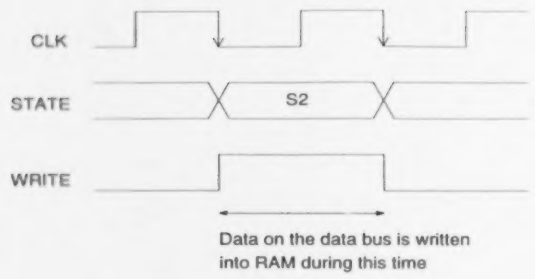


Figure 44: Timing Diagram

S0 until the START_BIST input is asserted by the external environment. When this signal is high, the BIST controller enters state S1 and begins executing the transparent test.

States S1, S2, S3 and S4 generate the $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march sequence. States S5 and S6 generate the background change sequence and S7 generates the $\Uparrow (r_b)$ march. In state S8, the background number is incremented and the SET_FAIL output is asserted if the signature computed during the background change sequence and the $\Uparrow (r_b)$ march differs from Signature 2. (The signature computed during the $\Uparrow (r_b w_b r_b w_b)$ march is compared to Signature 1 in S4.) State S9 is used to set or clear the PHASE2 latch, depending on the whether the test is at the end of phase 1 or phase 2. The signature analyzer is also cleared in State 9. In states S1, S2, ..., S9 the BIST_BUSY system output is asserted and the RAM is isolated by the multiplexers from the external environment.

In phase 1, the PHASE2 latch is cleared. During this phase, only states S1, S3, S5, S7 and S9 are traversed. S1 and S3 generate the read operations extracted from the $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march. S5 generates the reads from the background change sequences. S7 generates the $\Uparrow (r_b)$ march. S9 corresponds to the end of phase 1, where the PHASE2 latch is set and the signature analyzer is cleared. The BIST controller executes states S1 and S3 until the last address is reached, i.e. all the

read operations from the $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march in a test segment have been applied. The LOADREG1 signal is then asserted to store the signature computed during this period as Signature 1. Similarly, the LOADREG2 signal is asserted to store Signature 2 when the reads from the background change and the $\Uparrow (r_b)$ march are completed. The BIST controller then enters S9 where the signature analyzer is cleared using the CLRSIG signal. The PHASE2 latch is also set in S9 by asserting the LPHASE2 signal so that the phase 2 of the test segment will be executed.

In phase 2, the BIST controller executes the $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march in states S1, S2, S3 and S4. When this march is completed, the content of the signature analyzer is compared with Signature 1. If they differ, the SET_FAIL signal is asserted. Next, the background change sequence is applied to the RAM by states S5 and S6, followed by the final $\Uparrow (r_b)$ march in S7. The BIST controller then enters S8 where the content of the signature analyzer is compared to Signature 2. The SET_FAIL signal is again asserted if the signatures are different. At the same time, the background counter is incremented by asserting the NEXT_DBGR signal. Next, the BIST controller enters S9 and clears the PHASE2 latch and signature analyzer. Finally, the BIST controller returns to S1 to apply phase 1 for the next test segment.

When all test segments have been applied to the RAM, the BIST controller enters S10 and asserts the BIST_DONE system output and de-asserts BIST_BUSY. The BIST controller remains in this state until the system input BIST_ACK is asserted by the external environment. The controller then returns to S0 and the BIST RAM goes back to normal mode.

The VHDL code specifying the behavior of the BIST controller appears on pages 142 to 147 in Appendix A.

4.3 Data Path

The data path consists of the address generator, the background counter, the background code logic, the response analyzer, and the test pattern generator. Each hardware block will be discussed in detail in the next five subsections.

4.3.1 Address Generator

The address generator consists of a $(\log_2 n)$-bit Up-Counter, a Last Address detector and a D flip-flop with enable for latching the LAST_ADDR signal. The block diagram of the address generator is shown in Figure 45. The system clock

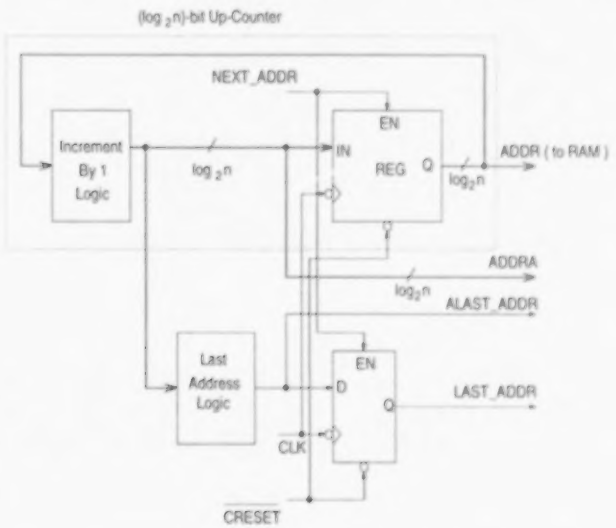


Figure 45: Address Generator

input provides the CLK signal to the flip-flops. The $\overline{\text{CRESET}}$ signal comes from the BIST controller. The $(\log_2 n)$-bit wide address bus ADDR is used to uniquely identify a bit in the $n \times 1$ RAM. The ADDRA address bus is the asynchronous counter outputs used by the BIST controller. ADDRA is equal to ADDR + 1 (mod n). The NEXT_ADDR signal acts as the counter enable input, i.e. when

NEXT_ADDR = 1, ADDR is incremented at the next falling clock edge by assigning ADDR to the current value of ADDRA. The ADDRA bus is fed into the Last Address Logic, which asserts ALAST_ADDR when the last value of ADDRA is reached in the count sequence, i.e. when ADDRA = $n-1$. The D flip-flop latches the ALAST_ADDR output to produce the LAST_ADDR output. Both the LAST_ADDR and ALAST_ADDR outputs are required by the BIST controller.

4.3.2 Background Counter

In this section, four different background counter designs will be described. The first design is the background counter for $V = 2$ and 3 shown in Figure 46.

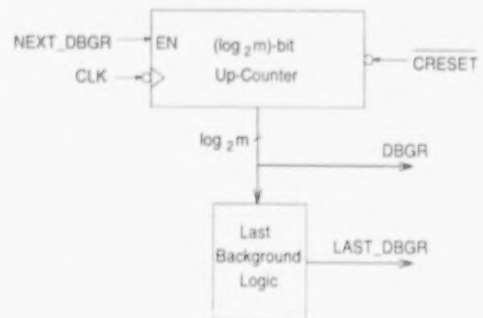


Figure 46: Background Counter for $V = 2$ and 3

The background counter consists of a $(\log_2)m$-bit Up-Counter and a block called "Last Background Logic". The NEXT_DBGR input enables the counter. The DBGR output from the counter is the background number which is used to select one background out of the m possible backgrounds as the current background. The Last Background Logic asserts the LAST_DBGR signal if the last background has been reached, i.e. when DBGR = $m-1$ (Note: DBGR counts from 0 to $m-1$).

The block diagram of the background counter for $V = 4$ is shown in Figure 47. The background number consists of two fields: DBGR (multiple bits) and DBGRC

(one bit). The $\overline{\text{CRESET}}$ signal clears both the DBGR and DBGRC outputs to 0. The background counter is enabled if the NEXT_DBGR input is asserted. The DBGR output from the $(\log_2 \frac{m}{2})$-bit up-counter increments one step through the range 0 to $\frac{m}{2} - 1$. When the DBGR output is at the maximum value $\frac{m}{2} - 1$ and NEXT_DBGR is asserted, the next falling clock edge sets the DBGRC output to 1 while the DBGR output goes back to 0. The DBGR output then will again increment in single steps through the range 0 to $\frac{m}{2} - 1$. The LAST_DBGR output is asserted when DBGRC = 1 and DBGR = $\frac{m}{2} - 1$. The corresponding two fields in the next background number are ADBGRC and ADBGR, where ADBGRC and ADBGR are the unlatched variants of DBGRC and DBGR respectively. The next background number is used for selecting the next background when determining the N bit values in the corresponding $(n, 3)$-exhaustive background matrix.

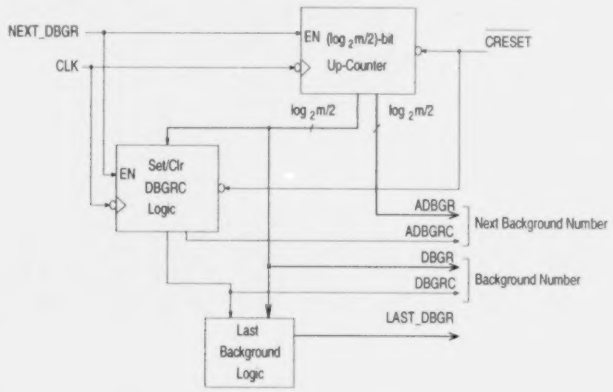


Figure 47: Background Counter for $V = 4$

The background number for $V = 5t$ has four fields: DBGR1, DBGR2, DBGRT and DBGRC as listed in Table 5. The DBGR1 field is $\log_2(1 + \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2})$ bits wide and increments from 0 to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$. The DBGR2 field is $\log_2(1 + \log_2 \sqrt{n})$ bits wide and increments from 0 to $\log_2 \sqrt{n}$. Both the DBGRT and DBGRC fields are 1-bit wide. The background number

Field	Width	Range
DBGR1	$\log_2(1 + \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2})$	$0, 1, \ldots, \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$
DBGR2	$\log_2(1 + \log_2 \sqrt{n})$	$0, 1, \ldots, \log_2 \sqrt{n}$
DBGRT	1	0, 1
DBGRC	1	0, 1

Table 5: Background Number for $V = 5t$

increments as follows:

for DBGRC = 0 to 1
  for DBGRT = 0 to 1
    for DBGR2 = 0 to $\log_2 \sqrt{n}$
      for DBGR1 = 0 to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$
        background number = | DBGRC | DBGRT | DBGR2 | DBGR1 |
      end
    end
  end
end

The block diagram of the background counter for $V = 5t$ is shown in Figure 48. The $\overline{\text{CRESET}}$ input is used to clear all four fields to 0. NEXT_DBGR is the background counter enable input. The block labeled "Set/Clr DBGRT Logic" toggles the DBGRT output from 0 to 1 or from 1 to 0 when DBGR1 and DBGR2 have reached their maximum values. Similarly, the Set/Clr DBGRC Logic toggles the DBGRC output when DBGRT is equal to 1 and both DBGR1 and DBGR2 have reached their maximum values. The LAST_DBGR output is asserted when DBGRC = 1, DBGRT = 1, DBGR2 = $\log_2 \sqrt{n}$ and DBGR1 =

$\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$. ADBGR1, ADBR2, ADBGRT and ADBGRC are the unlatched variants of DBGR1, DBGR2, DBGRT and DBGRC, respectively. These outputs select the next background that is used for determining the N bit values in the background matrix.

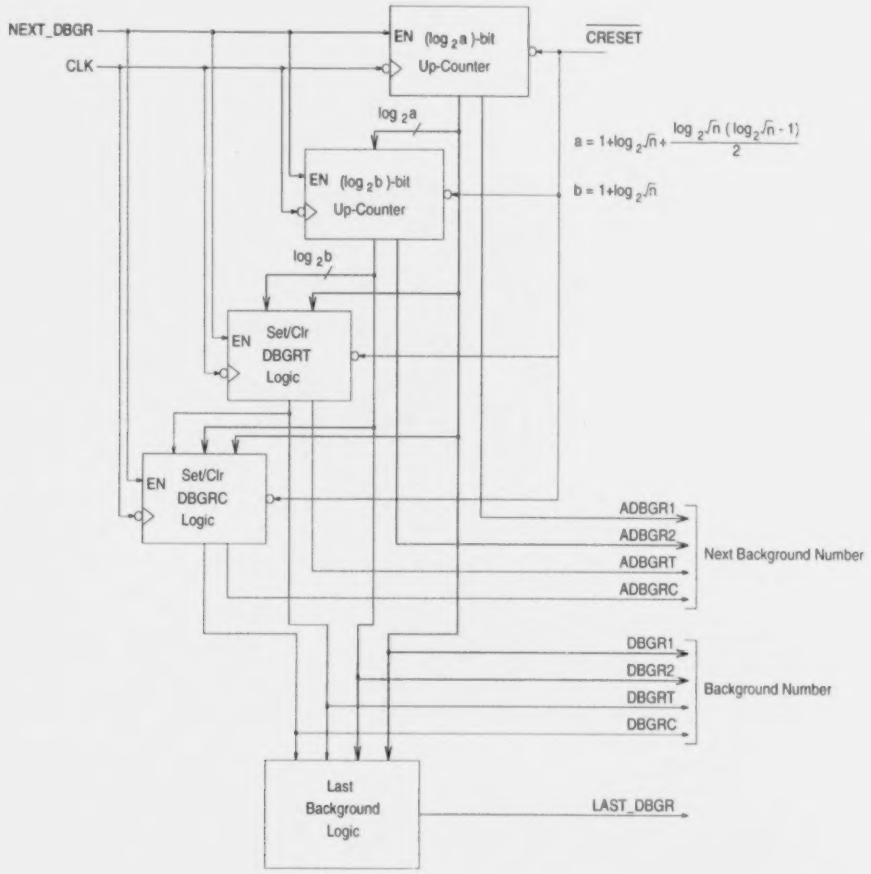


Figure 48: Background Counter for $V = 5t$

The three fields in the background number for $V = 5x$ (DBGR1, DBGR2 and DBGRC) are listed in Table 6. The DBGR1 and DBGR2 fields are $\log_2(1 + \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2})$ bits wide and increment independently from 0 to $\log_2 \sqrt{n} +$

Field	Width	Range
DBGR1	$\log_2(1 + \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2})$	$0, 1, \ldots, \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$
DBGR2	$\log_2(1 + \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2})$	$0, 1, \ldots, \log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$
DBGRC	1	0, 1

Table 6: Background Number for $V = 5x$

$\frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$. The DBGRC output is 1 bit wide.

The background number increments as follows:

for DBGRC = 0 to 1

for DBGR2 = 0 to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$

for DBGR1 = 0 to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$

background number = | DBGRC | DBGR2 | DBGR1 |

end

end

end

Figure 49 shows the block diagram of the background counter for $V = 5x$. The Set/Clr DBGRC Logic toggles the DBGRC output when DBGR1 and DBGR2 are both equal to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$. The LAST_DBGR output is asserted when DBGRC = 1 and both DBGR1 and DBGR2 are equal to $\log_2 \sqrt{n} + \frac{\log_2 \sqrt{n}(\log_2 \sqrt{n}-1)}{2}$. ADBGR1, ADBG2 and ADBGRC are the unlatched variants of DBGR1, DBGR2 and DBGRC, respectively.

In all four background counter designs, the background number rolls over to the first value after reaching the maximum values. Therefore when the background number is equal to the last background number, the next background number will be equal to the first background. This ensures that after the last background has

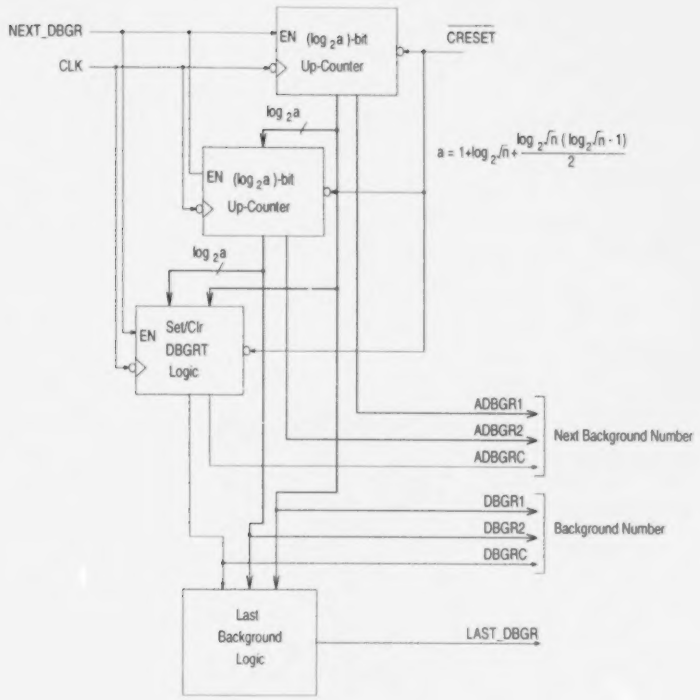


Figure 49: Background Counter for $V = 5x$

been applied to the memory, the final background change compares the P bits in the final background and the N bits in the first background to restore the original contents of the memory and ensures that the test is transparent.

4.3.3 Background Code Logic

The background code logic for $V = 3$ shown in Figure 50 generates the $(n, 2)$-exhaustive code described in section 3.2.1. The ADDRA input is the unlatched address bus from the address generator circuit. The background number DBGR comes from the background counter. AP and AN are the P and N bits corresponding to the unlatched ADDRA. The "Sumzero Logic" block counts the number of zeros in the ADDRA bus. Note that this is similar to Berger codes that are com-

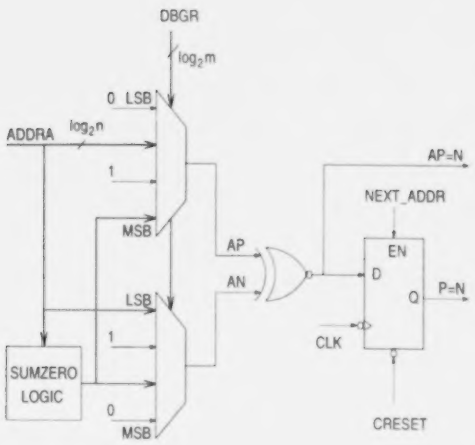


Figure 50: Background Code Logic for $V = 3$

monly used for error detection. However, the Sumzero Logic block cannot be used as a Berger code checker because the synthesized circuit for the Sumzero block may not necessary be totally self-checking. A $(\log_2 n + \log_2(\log_2 n + 1) + 2)$-input multiplexer is used to select the P bit based on the DBGR input. There are four sets of inputs to the multiplexer corresponding to the four fields in the $(n, 2)$-exhaustive code. The first set is a 1-bit wide 0 input corresponding to the all-zero column in the code. The second set is the ADDRA input. The third set is a 1 input. The output of the Sumzero Logic is connected to the fourth set of multiplexer inputs. The N bit is generated using a similar multiplexer but with all the inputs rotated down by one bit, i.e. the first set of inputs for the N bit multiplexer is ADDRA instead of a 0 input. The outputs of the multiplexers, AP and AN, are compared using an exclusive-NOR gate to produce the AP=N output. The AP=N signal is fed into a D flip-flop which generates the P=N signal.

Figure 51 shows the background code logic used to generate the $(n, 3)$-exhaustive code described in section 3.2.2. Two multiplexers are used to select two single bits which are fed into a 3-input XOR gate that produces the AP bit. The third in-

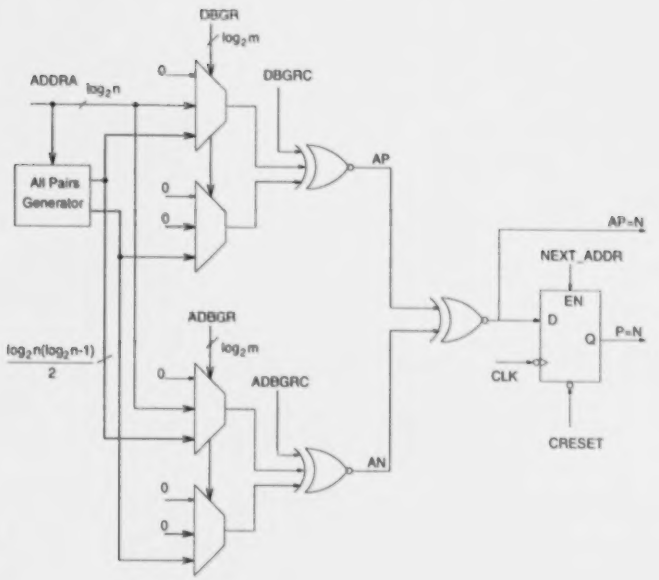


Figure 51: Background Code Logic for $V = 4$

put to the XOR gate is the DBGRC input. In Section 4.3.2, we have shown that DBGRC = 0 for the first $\frac{m}{2}$ backgrounds, and DBGRC = 1 for the last $\frac{m}{2}$ backgrounds. Therefore, when DBGRC = 0, the background code logic generates the AP bits in the first half of the code, and when DBGRC = 1 the second half of the code (which is equal to the complement of the first half) is generated. There are three sets of inputs to the multiplexers corresponding to the three fields in one half of the $(n, 3)$-exhaustive code. Similar to the background code logic for $V = 3$, the first set of inputs contains only the 0 input. The second set of inputs for the first multiplexer is the address bus ADDRA. The corresponding input to the second multiplexer is set to 0 so that when DBGRC = 0, i.e. in the first half, the AP bit is selected from the ADDRA bus; when DBGRC = 1, the AP bit is selected from the complement values of the bits in the ADDRA bus.

The "All Pairs Generator" block contains connections to all pair of lines in

the address ADDRA. There are two outputs from this block corresponding to the left and the right members of the address pairs. Each one of the outputs is a $\frac{\log_2 n(\log_2 n-1)}{2}$-bit wide bus. One output bus is connected to the third set of inputs to the first multiplexer. Similarly, the other output bus is connected to the second multiplexer. The third set of inputs generates the AP bits for the third field in the $(n,3)$-exhaustive code by XOR-ing all the pairs of bits from the address bus ADDRA. The circuit for generating the AN bits is similar to the circuit for the AP bits, but the next background number fields, ADBGRC and ADBGR, are used instead of DBGRC and DBGR.

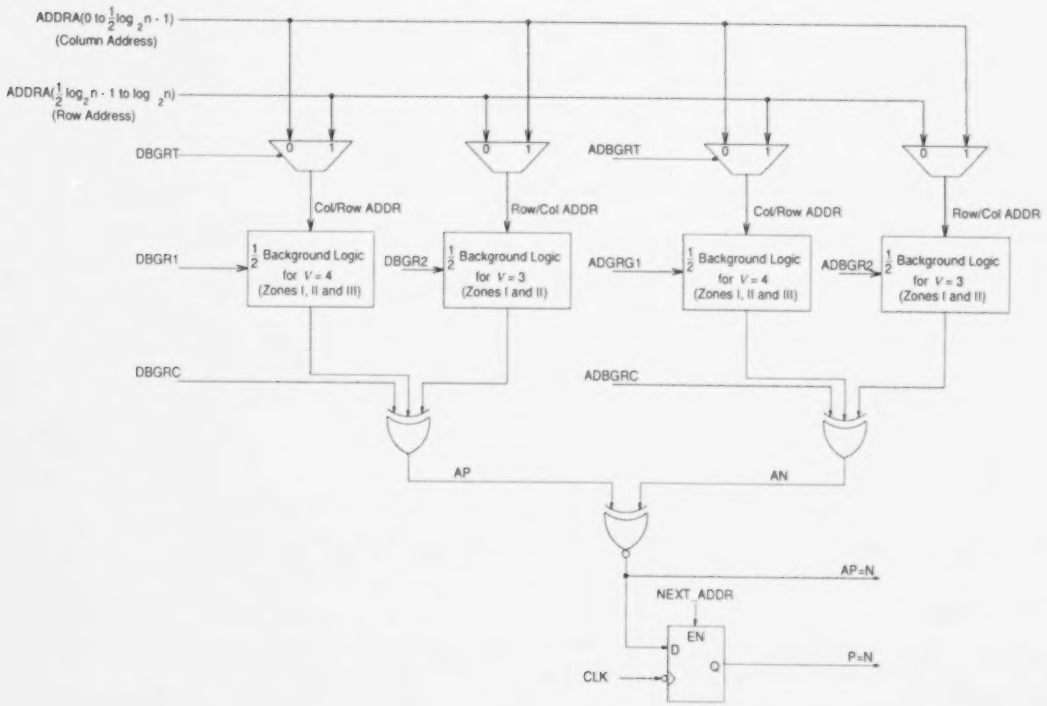


Figure 52: Background Code Logic for $V = 5t$

The background code logic for $V = 5t$ shown in Figure 52 generates the $(n,4)$-exhaustive code with respect to the scrambled T-neighborhoods described in sec-

tion 3.2.3. This code is used for detecting single active scrambled PNPSFs in an $n \times 1$ RAM where the storage cells are assumed to be arranged in a square $\sqrt{n} \times \sqrt{n}$ grid. The row address is composed of the higher order $\frac{\log_2 n}{2}$ bits of the address bus ADDRA, while the column address consists of the lower order $\frac{\log_2 n}{2}$ bits in ADDRA. The block labeled "$\frac{1}{2}$Background Logic for $V = 4$" refers to the background code logic that generates the first half of the corresponding $(\sqrt{n}, 3)$-exhaustive code. This half corresponds to Zone I, Zone II and Zone III (refer to Figure 18). The background code logic for generating this code is a slight modification of the circuit in Figure 51. The DBGRC signal is removed since only the first half code is required. The "$\frac{1}{2}$Background Logic for $V = 3$" block refers to the background code logic that generates the first half of the corresponding $(\sqrt{n}, 2)$-exhaustive code (Zone I and Zone II). The background logic in Figure 50 is modified by removing the Sumzero Logic block to obtain this logic.

In section 3.2.3, the construction process for the $(n, 4)$-exhaustive code with respect to T neighborhoods was described. The backgrounds obtained in Step 1 are generated by inputting the column address into the $\frac{1}{2}$Background Logic for $V = 4$ circuit to produce a $(\sqrt{n}, 3)$-exhaustive code. At the same time, the row address is input into the $\frac{1}{2}$Background Logic for $V = 3$ circuit to produce a $(\sqrt{n}, 2)$-exhaustive code. DBGRT and DBGRC are set to 0 in Step 1. DBGR1 selects an output bit in the $(\sqrt{n}, 3)$-exhaustive code and DBGR2 selects an output bit from the $(\sqrt{n}, 2)$-exhaustive code. The output bits from the two circuits are fed into a 3-input XOR gate. Since DBGRC is equal to 0, the AP bit is equal to the XOR of the two outputs from the $\frac{1}{2}$Background Logic for $V = 4$ circuit and the $\frac{1}{2}$Background Logic for $V = 3$ circuit. The matrices obtained in Step 2 are generated by setting DBGRT to 1. In this case, the row address and the column address are interchanged so that the row address goes into the $\frac{1}{2}$Background Logic for $V = 4$ while the column address goes into the $\frac{1}{2}$Background Logic for $V = 3$.

DBGRC is set to 1 to produce the matrices in Step 3 since the backgrounds in Step 3 are identical to the complement of the backgrounds obtained in Step 1 and Step 2. The circuit for generating the AN bits is similar to the circuit for the AP bits except the next background number (ADBGRC, ADBGRT, ADBGR2, ADBGR1) is used in place of (DBGRC, DBGRT, DBGR2, DBGR1).

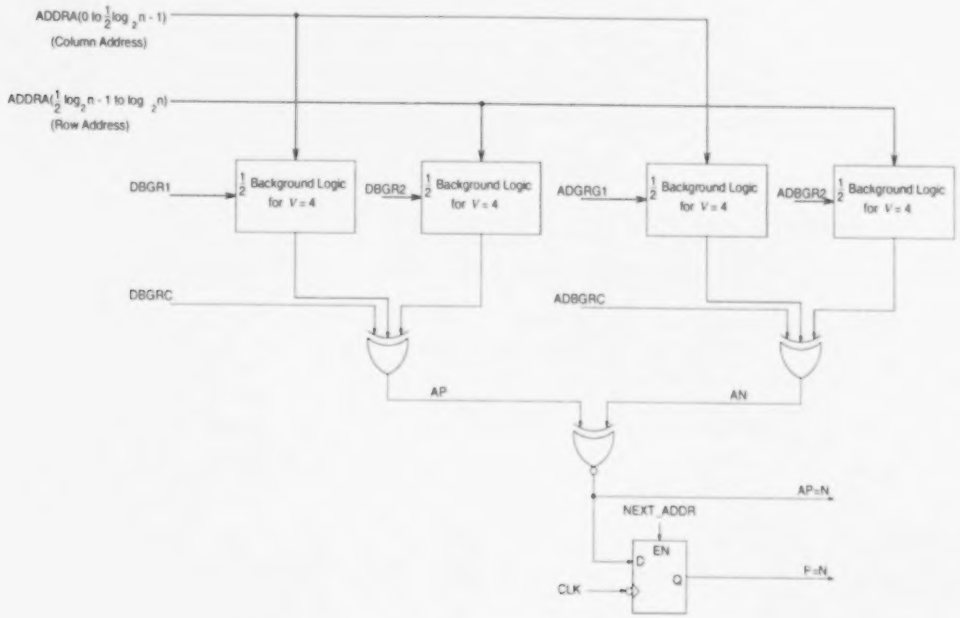


Figure 53: Background Code Logic for $V = 5x$

The background code logic for $V = 5x$ shown in Figure 53 is used to generate the $(n, 4)$-exhaustive code with respect to the scrambled neighborhoods of type 1. The background code logic is similar to the one for $V = 5t$ except that in this case, both the row and column addresses are input into two separate circuits that generate two sets of bits from a $(\sqrt{n}, 3)$-exhaustive code. DBGR1 and DBGR2 are used to select a bit out of each set. The output bits from the two circuits are then XOR-ed together with DBGRC. When DBGRC is 0, the background code

logic produces the backgrounds obtained in Step 1 of the code construction process (described in section 3.2.4). When DBGRC is equal to 1, the backgrounds in Step 2 are generated.

4.3.4 Response Analyzer

The block diagram for the response analyzer is shown in Figure 54. The response analyzer consists of an LFRS, two registers, a comparator, two multiplexers and a block labeled "Read Bit Flipper". Data from the RAM data bus is fed into the Read Bit Flipper and one of the multiplexers. In phase 1, the PHASE2 input is equal to 0, and the multiplexer selects the output from the Read Bit Flipper to sent to the LFSR. The Read Bit Flipper determines which bits have to be complemented before being input into the LFSR in this phase. Therefore, this block ensures that in the fault-free scenario, the signatures obtained in phase 1 will be the same as the ones in phase 2. In phase 2, the PHASE2 input is equal to 1, and the data from the RAM is sent directly to the LFSR. The enable input of the LFSR is connected to the READ input from the BIST controller so that the LFSR is only active during read operations. The CLRSIG input is used to clear the LFSR at the end of phase 1 and phase 2. The LOADREG1 and LOADREG2 inputs are used to enable Register 1 and Register 2. The output of the LFSR is input into the comparator. A multiplexer is used to select the source of the second input to the comparator. If the current state is S8, the output of the Register 2, Signature 2 is selected. Therefore, the comparator compares the content of the LFSR with Signature 2. If the current state is not equal to S8, Signature 1 is selected as the input to the comparator. The PASS output is set to 1 if the signatures are identical.

In section 3.5, we have shown that the transparent BIST scheme detects all sin-

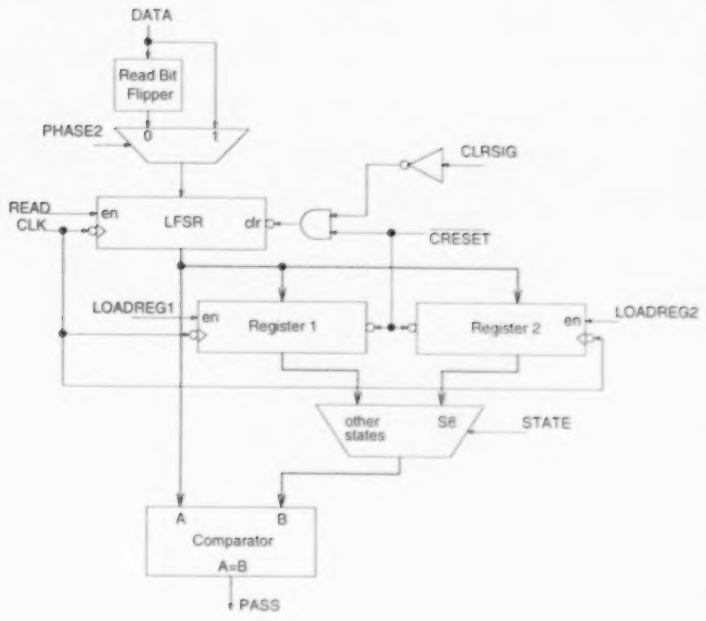


Figure 54: Response Analyzer

gle V-coupling faults if an LFSR with characteristic polynomial of degree $\lceil \log_2(n+1) \rceil + 2$ is used as the signature analyzer. Four different LFSRs are designed to satisfy this requirement. The VHDL description for the response analyzer is written in such a way so that an LFSR of degree 16 is instantiated for $n \leq 16383$ (16383 = 16K - 1). For $(16\text{K}-1) < n \leq (256\text{K}-1)$, an LFSR of degree 20 is instantiated. Similarly, LFSRs of degree 25 and 30 are instantiated for $(256\text{K}-1) < n \leq (8\text{M}-1)$ and $(8\text{M}-1) < n \leq (64\text{M})$, respectively.

4.3.5 Test Pattern Generator

Figure 55 shows the block diagram for the test pattern generator. The test pattern generator produces the data that is written into the RAM during the phase 2 of each test segment. The data from the RAM data bus is inverted because, in the transparent test algorithm, a memory cell is always written with the complement of the previously read data. A D flip-flop is used to store the inverted data and the BIST controller READ output is used to enable this flip flop, i.e. a new data bit is latched into the flip-flop every time a READ operation is executed. The output of the flip-flop is fed into a tri-state buffer. The buffer enable input is connected to the BIST controller WRITE signal so that the data in the flip-flop is driven onto the bi-directional data bus during a write operation.

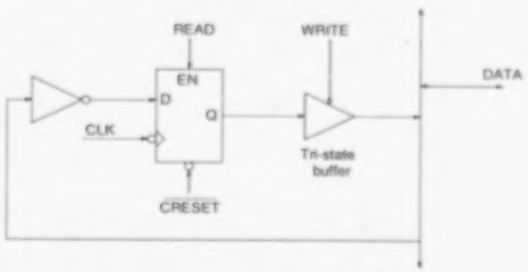


Figure 55: Test Pattern Generator

5 Design Evaluation

Section 5.1 of this chapter presents the results obtained from the logic simulations using the Verilog Logic Simulator in CADENCE 4.2.2. The cost of the BIST scheme in terms of area overhead and test time is discussed in section 5.2.

5.1 Simulation Results

The VHDL description of the BIST circuit is included in Appendix A. This description is input into the synthesis tools in Synopsys v3.1a, which then generates optimized BIST circuits using the CMOS4S technology. Complete BIST circuits can be synthesized based on only two input parameters n and V, where n is the RAM size and V correspond to the faults to be tested. V= 2, 3, 4, 15 or 25 correspond to single 2, 3, 4-coupling faults, scrambled active pattern sensitive faults and static or passive scrambled pattern sensitive faults, respectively. The BIST circuits produced by Synopsys are translated into EDIF output specifications, which in turn are imported into CADENCE 4.2.2 Design Framework II environment using the EDIFIN translator. The circuit schematics in Design Framework II are then verified using the Verilog Logic Simulator. An example of the top-level schematic of the BIST circuit for $n = 4$ and $V = 3$ is shown in Figure 56.

In this section, we discuss the results of two simulations for the BIST circuit shown in Figure 56. The first simulation uses a fault-free 4-cell RAM; the output waveforms are shown in Figures 68 to 74 in Appendix C. There are four inputs: the system clock CLK, the system reset CLR, the START_BIST input and the BIST_ACK input. The circuit outputs are BIST_BUSY, BIST_DONE and PASS/$\overline{\text{FAIL}}$. The BIST controller READ and WRITE outputs are shown to indicate the type of operations that are performed on the RAM during the application of the self-test. Note that the first eight operations in Figure 68 are two reads to

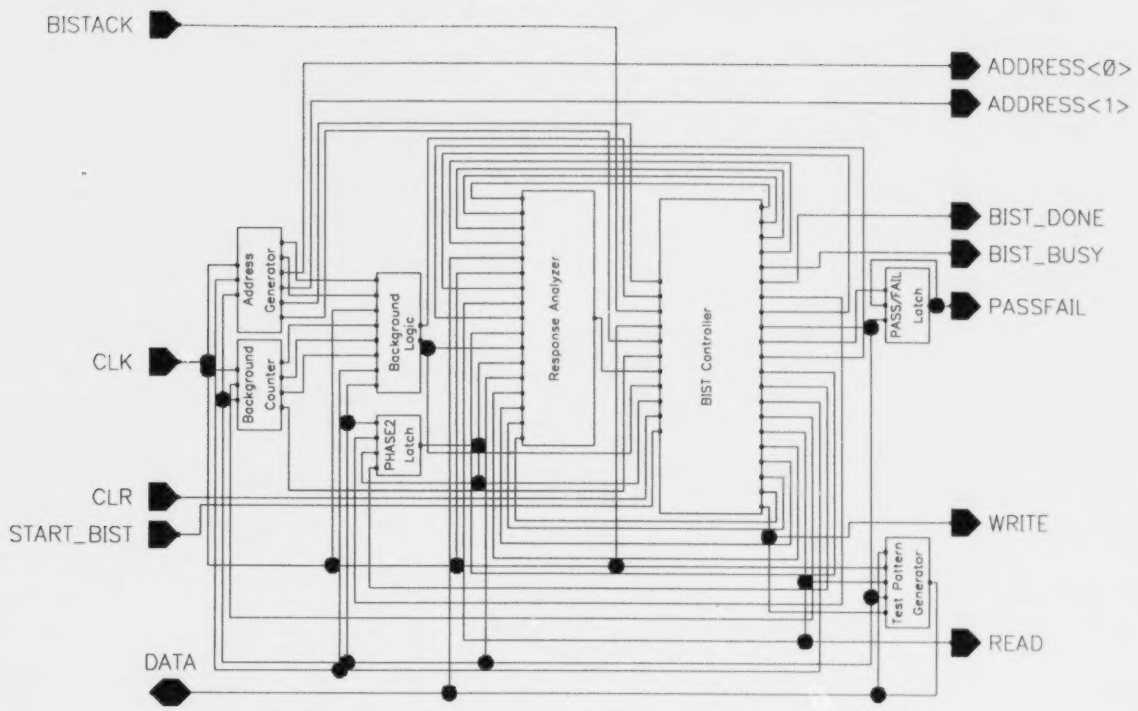


Figure 56: BIST Circuit Schematic

each of cells 0, 1, 2, and 3. Similarly, the ADDR output is shown to indicate which cell is being accessed. The contents of the 4-cell RAM, MEM<0>, MEM<1>, MEM<2> and MEM<3> are also shown. In Figure 68 the initial contents of cells 0, 1, 2 and 3 are the values 0, 1, 1, and 0, respectively. The system clock is set at 10 MHz. At time t=100ns, the active low system reset signal CLR is asserted to 0. This signal initializes the BIST controller to state S0, where the non-RAM flip-flops in the BIST circuit are cleared using the BIST controller $\overline{\text{CRESET}}$. This includes the flip-flops for the READ, WRITE, BIST_BUSY, BIST_DONE, and the ADDR bus. At the same time the PASS/$\overline{\text{FAIL}}$ output is set to 1, indicating PASS

since no faults have been detected yet. In this simulation, we use the CLR signal to initialize the contents of the RAM to 0, 1, 1, 0. In a real BIST RAM, the content of the RAM would not be initialized at this stage. The CLR input is set back to 1 (i.e. de-asserted) at t=200ns. At t=300ns the START_BIST input is asserted to activate the self-test routine. The BIST RAM enters the test mode at the next negative clock edge at t=400ns and BIST_BUSY is asserted. At this time the READ signal is asserted to produce the first read operation at ADDR 0 in phase 1 of segment 1. The READ signal remains high at the next clock period indicating that cell MEM<0> is read again. This read operation corresponds to the second read operation extracted from the first $(r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march element that is applied to MEM<0>. At the next negative clock edge, ADDR is incremented to 1 while the READ signal remains asserted for two clock periods, indicating that cell MEM<1> is being read twice. The double read operations are repeated for cells MEM<2> and MEM<3>.

Once the read operations extracted from the $\Uparrow (r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march have been applied to the RAM, the corresponding read operations from the background change are performed. The (4,2)-exhaustive background matrix for $n = 4$ and $V = 3$ is shown in Figure 57. Since the P bit and the N bit for cell MEM<0> are the same in the first background, no read operation is performed on this cell and the ADDR is incremented to 1. Similarly, no read operation is applied to cell MEM<1>. Both cells MEM<2> and MEM<3> are read since their P bits are not the same as their N bits.

0	000110
1	001101
2	010101
3	011100

Figure 57: (4,2)-exhaustive Code

Next, starting at t=1600ns, all the cells are read once in ascending address order. These read operations correspond to the final $\Uparrow (r_b)$ at end of phase 1 in segment 1. The next clock period (2000ns to 2100 ns) is used to clear the signature analyzer and set the PHASE2 latch.

Phase 2 of segment 1 starts at t=2100ns with a read operation to cell MEM<0>. In the next three clock periods, MEM<0> is written to 1, read a second time and then written back to zero. The same $(r_b w_{\bar{b}} r_{\bar{b}} w_b)$ operations are applied in turn to cells MEM<1>, MEM<2> and MEM<3>, where the b values for the three cells are 1, 1 and 0, respectively. Next, the background change operations are executed beginning at t=3700ns. No background operations are applied to cells MEM<0> and MEM<1> and the two clock periods from t=3700 to t=3900ns are used to increment the address ADDR to 2. The read signal is then asserted indicating that cell MEM<2> is read. This is followed by a write operation which changes the content of MEM<2> from 1 to 0. Cell MEM<3> undergoes similar operations except that its contents is changed from 0 to 1. The $\Uparrow (r_b)$ march is then applied to all the cells. Once this march is complete, two clock periods from t=4700 to t=4900 are used to increment the background number (state S8 in state diagram), and then clear the signature analyzer and set the PHASE2 latch (state S9). The operations of the next test segment are now applied to the RAM.

The last test segment in this first simulation finished at time t=27500 (as shown in Figure 74). At this time, the BIST_BUSY is de-asserted while the BIST_DONE is asserted. Since, a fault-free RAM is used in the simulation, no faults were detected and the PASS/$\overline{\text{FAIL}}$ signal remains high at this time. (Note that a stuck-at fault that keeps the PASS/$\overline{\text{FAIL}}$ line high can be detected using the two additional input signals; one to force the PASS/$\overline{\text{FAIL}}$ line to low and the other to force it to high.) The BIST_DONE signal stayed high until the BIST_ACK signal is received at time t=28500ns. At the next negative clock edge, the BIST_DONE

is de-asserted and the BIST RAM returns to normal mode. Note that the test is transparent because the contents of the RAM at the end of the test are the same as the initial contents.

The second simulation is performed using a faulty RAM. The RAM is assumed to have a 3-coupling fault where an ↓MEM<2> causes ↓MEM<3> when MEM<0> contains a 1. The output waveforms for this simulations are shown in Figure 75 to 81 in Appendix C. For the 3-coupling fault to be triggered, MEM<0>MEM<3> = 11, and MEM<2> has to be written from 1 to 0. This event first occurs in segment 3 (Figure 77) during the background change from background number 2 to background number 3 (column 3 to column 4 in the background matrix in Figure 57). Since the initial content of MEM<2> is equal to 1, all the 0 entries in the background matrix for MEM<2> correspond to the initial value 1. Therefore, when MEM<2> undergoes a background change with its P bit equal to 0 and its N bit equal to 1, the content of the cell actually changes from 1 to 0. Since MEM<0>MEM<3> = 11 at the time, the ↓ MEM<2> causes the 3-coupling fault to be triggered. The content of the signature analyzer is compared with Signature 2 at t=13700ns and the PASS/$\overline{\text{FAIL}}$ is set to 0. Note that the fault is triggered again during the $\Uparrow (r_b w_b r_b w_b)$ in segment 5 (Figure 79).

5.2 Results of Layout Experiments

Using the AutoLayout tools in CADENCE 4.2.2, the layouts of the BIST circuits were produced using BNR's CMOS4S technology with 1.2μm design rules. The RAM circuit has been omitted and only the area occupied by the BIST circuitry was measured. We ignored the area occupied by the input and output pads in the area measurements because in the BIST scheme, both the BIST circuit and the RAM are contained in the same chip (see Figure 58). The BIST circuit area

measurements are used to estimate the area overhead costs of the BIST scheme. Table 7 lists the area occupied by the BIST circuits, the area of the RAM and the percentage of area overhead for $V = 3$ and different RAM sizes. The information on the RAM area is obtained from the Proceedings of the IEEE International Solid State Circuits Conference. The percentage of area overhead is calculated as $\frac{\text{BIST area}}{\text{RAM area}} \times 100\%$.

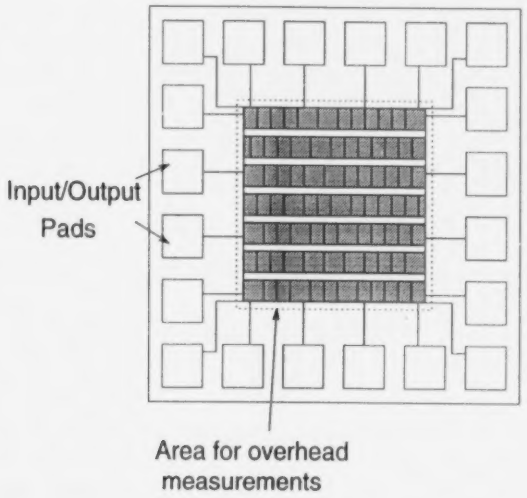


Figure 58: Area Used for BIST Overhead Approximations

Figure 59 shows the BIST circuit area overhead for $V = 3$, 4, $5t$ and $5x$. The area overhead decreases rapidly as the RAM size increases. This is because the increase in the size of the BIST circuit is relatively slow compared to the increase in the size of the memory circuit. For example, the BIST controller circuit is independent of the RAM size, and therefore its size does not increase with the increase in RAM size. Also we only require one additional bit in the address bus when the memory size is doubled. Therefore the address generator will increase by an additional flip-flop and a few logic gates while the number of memory cells in the RAM is doubled. The area overheads for $V = 4$ are higher than the ones for $V = 3$

RAM size ($n \times 1$)	RAM area (mm^2)	BIST area (mm^2)	% overhead
64k	10.098	2.1370	21.16%
256k	40.39	2.2839	5.65%
1M	161.56	2.3802	1.47%
4M	646.24	2.4806	0.38%
16M	2584.96	2.6369	0.10%

Table 7: BIST Area Overhead for $V = 3$ and Typical Sizes of RAM

because the background code logic for $V = 4$ is more complex than the background code logic for $V = 3$. Also, the background counter circuit for $V = 4$ is slightly bigger than the circuit for $V = 3$. Note that the rest of the BIST circuits for $V = 4$ (address generator, BIST controller, response analyzer and test pattern generator) are identical to the circuits for $V = 3$. The area overheads for $V = 4$ is higher than the overheads for $V = 5t$ and $V = 5x$. This is because the background code logic for $V = 4$ requires $1 + \log_2 n + \frac{\log_2 n(\log_2 n - 1)}{2}$-input multiplexers (see Figure 50). The background code logics for $V = 5t$ and $V = 5x$ however, only require $1 + \frac{1}{2}\log_2 n + \frac{\log_2 n(\frac{1}{2}\log_2 n - 1)}{4}$-inputs multiplexers. For example, when n is equal to 1M, the background code logic for $V = 4$ will require 211-input multiplexers while the background code logics for $V = 5t$ and $V = 5x$ only contain 56-input multiplexers.

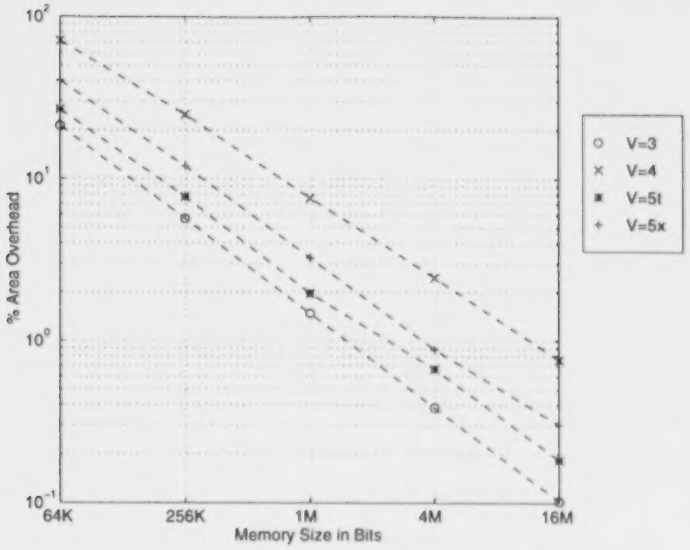


Figure 59: Area Overhead

6 Summary and Conclusions

In this thesis, a tool for automatically generating BIST circuits was presented which can be used for detecting single V-coupling faults in embedded RAMs. The BIST circuits implement a transparent self-test that can be used for detecting single 2, 3 and 4-coupling faults, as well as active, passive and static scrambled 5-cell NPSFs.

Chapter 2 presented background material in memory testing and circuit synthesis. In Chapter 3, we developed a BIST scheme based on the deterministic test for detecting single V-coupling faults proposed in [22]. The structure of the deterministic test was described in section 3.1. A background matrix is used for generating the test patterns to be applied to the RAM under test. To detect all single V-coupling faults, the rows of the background matrix have to form an

$(n, V-1)$-exhaustive code. Descriptions of easily generated $(n, V-1)$-exhaustive codes for $V = 3$, 4, $5t$ and $5x$ were presented. A technique described in [12] is used to transform the deterministic test into a transparent test. The technique essentially ensures that when the transparent test is applied to a RAM, each cell in the RAM is complemented an even number of times. Therefore, the content of each cell will be the same as its initial content if no fault is detected in the RAM. Also, the response of the RAM is compacted using signature analysis, where an LFSR is used as the signature analyzer. Signature analysis is an efficient and low-cost data compaction technique. However, like any other data compaction methods, signature analysis is vulnerable to the possibility of aliasing. Aliasing occurs when a faulty RAM produces a response sequence that is different from that of a fault-free RAM, but the resulting compacted binary sequences or signatures are identical.

In order to reduce the probability of aliasing, we used the technique described in [10]. This technique modifies the transparent test to obtain an aliasing-free BIST scheme if no more than two errors are created at a time. The test is modified so that, instead of comparing two signatures once, different signature pairs are computed and compared many times. The resulting aliasing-free test is divided into segments where each segment corresponds to a background in the background matrix. Four signatures are computed in each test segment, two in the signature generating phase (phase 1) and two during the application of the test (phase 2). A signature comparison is performed after an interval that could possibly introduce no more that two errors as a result of one fault. If multiple faults occur in the RAM, the probability that these faults will escape detection was still found to be very low. For example, the probability that two single 3-coupling faults present in a $1M$ RAM will not be detected is in the order of 10^{-12}.

In Chapter 4, the BIST RAM design was presented in detail. The BIST RAM has two operating modes: normal mode and self-test mode. In normal mode, the

$n \times 1$ RAM under test performs normal read and write operations requested by the external environment. In self-test mode, the RAM is isolated from the external environment and the transparent test is applied to the RAM. The BIST RAM is divided into three main blocks: the $n \times 1$ RAM, the BIST controller and the data path. The circuit implementation and functionality of the BIST controller and the data path blocks were described in detail in sections 4.2 and 4.3. The VHDL descriptions for the BIST circuitry are included in Appendix A.

BIST circuits can be automatically generated using the synthesis tools in Synopsys and the layout tools in CADENCE 4.2.2. The synthesizable transparent BIST scheme has only two input parameters, n and V. Parameter n is the size of the RAM under test. V corresponds to the type of fault to be detected. Chapter 5 presented the results from simulations and layout experiments. Two logic simulations using the Verilog Logic simulator in Cadence 4.2.2 were analyzed. The first simulation was performed on synthesized BIST circuits with $n = 4$ and $V = 3$. We used a fault-free 4×1 RAM in this simulation to verify the BIST scheme. The second simulation was performed using the same BIST circuit but with a faulty RAM which contains a 3-coupling fault. We showed that the fault was detected and that the PASS/$\overline{\text{FAIL}}$ output was set to low indicating that the RAM was faulty. In Section 5.2, the BIST circuit design was evaluated in terms of the percentage of area overhead. For a 1M RAM, the area overhead for V=3, 4, $5t$ and $5x$ were estimated to be 1.47%, 7.5%, 1.96% and 3.24%, respectively. It is worth noting here that the overhead can be further reduced if the RAM under test is partitioned into many equally-sized subarrays, as is the case in large commodity DRAMs . In such a situation the subarrays could be tested in parallel and could share one copy of much of the BIST circuits. Each subarray would only need its own copy of the test pattern generator and the read bit flipper (in the response analyzer). A Multiple-Input Signature Register ($MISR$) could be used as the common signa-

ture analyzer that is shared by all the subarrays. In phase 1 of a test segment, the inputs to the MISR could consist of the outputs from the read bit flipper which are XOR-ed together. In phase 2, the data bits from the subarrays could be XOR-ed together and fed in to the MISR. For example, we would require 32 copies of the test pattern generator and the read bit flipper circuit to test 32 subarrays in parallel. The 32 output bits from the read bit flipper could be XOR-ed together to form a 16-bit output that could be fed into the MISR. Similarly, the 32 data bits could be XOR-ed together to form the 16-bit input to the MISR in phase 2 of a test segment. However we can only have at most one fault in the RAM for the scheme to be aliasing-free since the signature analyzer is common to all the subarrays. But we have already shown that even if multiple faults occur in the RAM, the probability that the faults will not be detected was found to be very low.

Table 8 lists the overhead required some of the BIST RAMs that have been proposed in literature [7, 26, 27, 28, 29, 30]. We use the acronyms SAF, CF and NPSF to represent stuck-at faults, 2-cell coupling faults and neighborhood pattern-sensitive faults, respectively.

It is difficult to compare our BIST RAM scheme with the BIST RAM schemes listed in Table 8 since the fault coverage is different in each scheme [7]. Generally the area overhead of our BIST scheme is acceptable especially for RAM sizes $n \geq 4$M. For instance, the overhead for our scheme is $< 1\%$ ($V = 3$, $5t$ and $5x$) for $n = 4$M, which is the same as the overhead reported for the CMOS dynamic RAM with BIST scheme. However, our scheme has better fault coverage since the CMOS dynamic RAM with BIST which implements the well-known checkerboard test [7] does not detect all stuck-at faults. In addition to single 2, 3, and 4-coupling faults, as well as active, passive and static scrambled 5-cell pattern-sensitive faults, our BIST RAM scheme also detects the simple stuck-at faults.

Implementation	Faults Detected	Overhead
On-chip compact test scheme [26]	SAFs, CFs 5-cell static NPSFs	1.21% (n=64K) 0.09% (n=1M)
Parallel test using signature analyzer [27]	SAFs, some CFs	0.4% (n=256K)
Parallel test for VLSI memories [28]	SAFs, some CFs	< 1% (n=2M)
Parallel test for pattern-sensitive faults [29]	static & dynamic 9-cell NPSFs	0.4% (n=256K)
CMOS dynamic RAM with BIST [30]	some SAFs (does not detect decoder faults)	< 1% (n=4M)

Table 8: Area Overhead of Some Proposed BIST RAMs

Table 9 lists the typical test time assuming a RAM cycle time of 100 ns.

Note that the test times also can be reduced by a factor of k if the RAM is partitioned into $k > 1$ equally-sized subarrays that could be tested in parallel. Test times typically should be under one minute. The mimimun values for k required in order to reduce the test times for 16M RAMs to less than one minute is tabulated in Table 10. It can be seen that the BIST RAM scheme can test a 16M RAM for static and passive scrambled pattern-sensitive faults in approximately 31.6 seconds if the RAM is partitioned into 4096 4K subarrays.

Another advantage of our BIST RAM is that the scheme does not require any modification to the decoder, read/write logic, sense amplifier circuits, or the memory cell array. In the BIST RAM normal mode the performance penalty due to the additional multiplexers, which appear in the path from the address, read and write inputs from the external environment to the RAM, is negligible.

V	Test Time	
	n = 1M	n = 16M
2	2.6 s	41.9 s
3	35.4 s	9.4 min
4	9.2 min	2.4 hr
$5t$	48.1 min	12.8 hr
$5x$	2.3 hrs	36.0 hr

Table 9: Typical Test Time

V	Number of Subarrays k	Subarray Size	Test Time
2	No partitioning required	-	41.9 s
3	16	1M	35.3 s
4	256	64K	34.1 s
$5t$	1024	16K	45.0 s
$5x$	4096	4K	31.6 s

Table 10: Test Time for 16M RAM After Partitioning

The BIST RAM scheme is vulnerable to stuck-at high faults that affect the PASS/$\overline{\text{FAIL}}$ output. In the presence of these faults, the PASS/$\overline{\text{FAIL}}$ output would always indicate that a RAM has passed the BIST routine even if faults were detected. One possible solution is to test for stuck-at faults that can occur along the signal path from the comparator output (in the response analyzer) to the PASS/$\overline{\text{FAIL}}$ output to the external environment using an XOR gate (see Figure 60). An additional input Test_PF is used to toggle the PASS input into the BIST Controller. Test_PF could then be used to deliberately make the self-test indicate

a failure. If PASS=0, then SET_FAIL will be asserted which in turn would clear the PASS/$\overline{\text{FAIL}}$ latch. If PASS=1, then SET_PASS will be asserted to set the PASS/$\overline{\text{FAIL}}$ output to high.

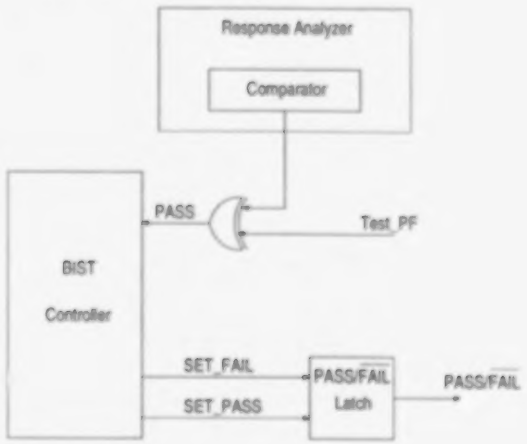


Figure 60: Test for Stuck-at Faults affecting the PASS/$\overline{\text{FAIL}}$ Output

As far as future work is concerned, it would be interesting to modify the BIST RAM design to detect single V-coupling faults in $n \times k$ RAMs. We would assume that V-coupling faults could involve cells from different k-bit words as well as multiple cells within the same words, in any combination. The rows of the background matrix used in this case would have to form an $(n \times k, V-1)$-exhaustive code. The $(r_b w_{\bar{b}} r_{\bar{b}} w_b)$ march element would be applied to each bit in a word. Instead of comparing a single P bit with a single N bit during a background change, the P bits for all the k cells in a word would be compared to the corresponding N bits for the k cells in the same word. This is due to the fact that in a $n \times k$ RAM, a read or write operation involves k cells. Therefore, all the P bits and the N bits for a word in the RAM would have to collected before a background change is applied to the word. One possible approach would be to have an additional state in the BIST controller in which the P and N bits for the k cells in a word could be stored

in a register. The address counter would have to be modified so that it consists of two counters, a word counter and a bit counter. The word counter would point to the current memory word, while the bit counter would point to the current bit in the word. The LAST_BIT status signal would be used to indicate if k number of P or N bits have been collected. The P and N bits are generated in parallel and stored in two shift registers. The BIST controller would stay in this additional state until the LAST_BIT signal is asserted. The background logic would assert the $P = N$ signal only if the P bits for all k cells in a word are the same as the N bits for the RAM word. The two shift registers would provide the test pattern to be applied to the word during the background change.

A further open problem is to design an easily-generated $(n, 4)$-exhaustive code. This code could be used for detecting single 5-coupling faults involving any five cells in the memory. One advantage of using such a code is that no assumptions would have to be made regarding the logical to physical mapping of the cell addresses and therefore this code could be used for testing all possible 5-cell pattern sensitive faults. However, the test times for such a code would be expected to be very long.

References

[1] B. Prince, *Semiconductor Memories.* Chichester, U.K.: John Wiley Sons, 1991.

[2] J. Robert J. Feugate and S. M. McIntyre, *Introduction to VLSI Testing.* Englewood Cliffs, New Jersey, U.S.A.: Prentice Hall, 1988.

[3] B. Prince and G. Due-Gundersen, *Semiconductor Memories.* Chichester, U.K.: John Wiley Sons, 1983.

[4] K. Tsukamoto and H. Morimoto, "Process and Device Technologies for Subhalf-Micron LSI Memory," *IEICE Transactions on Electronics*, vol. E77-C, pp. 1343–1349, August 1994.

[5] M. Horiguchi et al., "An Experimental 220Mhz 1Gb DRAM," in *Proceedings 1995 IEEE International Solid-State Circuits Conference*, pp. 252–253, 1995.

[6] T. Sugibayashi et al., "A 1Gb DRAM for File Applications," in *Proceedings 1995 IEEE International Solid-State Circuits Conference*, pp. 254–255, 1995.

[7] M. Franklin and K. K. Saluja, "Built-in Self-Testing of Random-Access Memories," *IEEE Computer*, vol. 23, pp. 17–26, October 1990.

[8] A. J. van der Goor, *Testing Semiconductor Memories: Theory and Practice.* Chichester, U.K.: John Wiley Sons, 1992.

[9] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design.* New York, N.Y., U.S.A.: Computer Science Press, 1990.

[10] V. N. Yarmolik, M. Nicolaidis, and O. Kebichi, "Aliasing-Free Signature Analysis for RAM BIST," in *Proceedings 1994 International Test Conference*, pp. 368–377, 1994.

[11] B. Koeneman in *Proceedings 1986 IEEE Design for Testability Workshop*, (Vail, Colorado, U.S.A), Apr. 26 - May 2 1986.

[12] M. Nicolaidis, "Transparent BIST for RAMs," in *Proceedings 1992 International Test Conference*, (Baltimore, MD, U.S.A.), pp. 598–607, Sept. 20-24 1992. IEEE Comp. Soc., Washington, 1992.

[13] H. D. Oberle, M. Maue, and P. Muhmenthaler, "Enhanced Fault Modeling for DRAM Test and Analysis," in *Digest of IEEE VLSI Test Symposium*, (Atlantic City, NJ, U.S.A.), pp. 149–154, April 1991.

[14] R. Nair, S. M. Thatte, and J. A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories," *IEEE Transactions on Computers*, vol. C-27, pp. 572–576, June 1978.

[15] A. J. van de Goor and C. A. Verruijt, "An Overview of Deterministic Functional RAM Chip Testing," *ACM Computing Surveys*, vol. 22, pp. 5–33, March 1990.

[16] J. P. Hayes, "Detection of Pattern Sensitive Faults in Random Access Memories," *IEEE Transactions on Computers*, vol. C-24, pp. 150–157, February 1975.

[17] A. Tuszynski, "Memory Testing," in *VLSI Testing* (T. W. Williams, ed.), Elsevier Science Publishers B.V., Amsterdam, 1986.

[18] M. Franklin and K. K. Saluja, "An Algorithm to Test RAMs for Physical Neighborhood Pattern Sensitive Faults," in *Proceedings 1991 International Test Conference*, pp. 675–684, 1991.

[19] K. Kinoshita and K. K. Saluja, "Built-in Testing of Memory Using an On-Chip Compact Testing Scheme," *IEEE Transactions on Computers*, vol. 35, pp. 862–870, October 1986.

[20] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis.* New York, U.S.A.: McGraw-Hill, 1994.

[21] R. Lipsett, C. F. Schaefer, and C. Ussery, *VHDL: Hardware Description and Designs.* Boston U.S.A: Kluwer Academic Publishers, 1993.

[22] B. F. Cockburn, "Deterministic tests for Detecting Single V-Coupling Faults in RAMs," *Journal of Electronic Testing: Theory and Applications*, vol. 5, pp. 91–113, 1994.

[23] K. K. Saluja and K. Kinoshita, "Test Pattern Generation for API Faults in RAM," *IEEE Transactions on Computers*, vol. C-34, pp. 284–287, March 1985.

[24] B. F. Cockburn and N. Y. Sat, "A Transparent Built-In Self-Test Scheme for Detecting Single V-Coupling Faults in RAMs," in *Proceedings 1994 IEEE International Workshop on Memory Technology, Design and Testing*, pp. 119–124, 1994.

[25] B. F. Cockburn and N. Y. Sat, "Synthesized Transparent BIST for Detecting Pattern-Sensitive Faults in RAMs." to appear in the Proceedings of the 1995 IEEE International Test Conference.

[26] K. K. Saluja, S. H. Sng, and K. Kinoshita, "Built-in Self-Testing RAM: A Practical Alternative," *IEEE Design and Test of Computers*, vol. 4, pp. 42–51, February 1987.

[27] T. Sridhar, "New Parallel Test Approach for Large Memories," in *Proceedings 1985 International Test Conference*, (Los Alamitos, CA, U.S.A.), pp. 462–470, Computer Society Press, 1985.

[28] J. Inoue et. al., "Parallel Testing Technology for VLSI Memories," in *Proceedings 1985 International Test Conference*, (Los Alamitos, CA, U.S.A.), pp. 1066–1071, Computer Society Press, 1985.

[29] P. Mazumder and J. Patel, " Parallel Testing for Pattern-Sensitive Faults in Semiconductor Random-Access Memories," *IEEE Transactions on Computers*, vol. 38, pp. 394–407, March 1989.

[30] T. Ohsawa et al, "A 60-ns 4-Mbit CMOS DRAM with Built-in Self-Test Function," *IEEE Journal of Solid-State Circuits*, vol. 22, pp. 663–668, October 1987.

A VHDL Code

Header File

File: header.vhd

```
-- This header file contains three libraries:
-- CONVERT_TYPE contains type conversion functions not found in the standard
-- IEEE libraries
-- BTYPES contains the STATES type declaration
-- FIND_PARAMETERS contains functions to determine:
--      M, width address bus
--      S, number of bits to represent the number of zeros in address bus
--      NUMXOR, number of pairs of address lines
--      NUMDBGR, number of backgrounds
--      NUMDBGR1, number of backgrounds in base matrix
--      NUMDBGR2, number of backgrounds in Zones I and II of base matrix
--      LFSRDEG, the degree of LFSR

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.MATH_REAL.all;

-- Library CONVERT_TYPE contains 3 functions:
-- INTEGER_TO_BIT_VECTOR: converts an integer to type BIT_VECTOR
-- CONVERT_TO_X01Z: converts a BIT to X01Z (to set the output of test pattern
--                      generator to high impedence)
-- CONVERT_TO_BIT : converts a X01Z bit to type BIT

package CONVERT_TYPE is
        function INTEGER_TO_BIT_VECTOR(X,Y: INTEGER)
        return BIT_VECTOR;
        function CONVERT_TO_X01Z(X: BIT)
        return X01Z;
        function CONVERT_TO_BIT(X: X01Z)
        return BIT;
end CONVERT_TYPE;
package body CONVERT_TYPE is
        function INTEGER_TO_BIT_VECTOR(X,Y : INTEGER)
        return BIT_VECTOR is
                variable T : INTEGER;
                variable J : BIT_VECTOR(Y-1 downto 0);
                variable K : UNSIGNED(Y-1 downto 0);
                begin
                        K := CONV_UNSIGNED(X,Y);
                        for I in 0 to Y-1 loop
                                if K(I)= '0' then J(I) := '0';
                                else  J(I) := '1';
                                end if;
                        end loop;
                        return J;
                end;
        function CONVERT_TO_X01Z(X: BIT)
        return X01Z is
                begin
                        if X = '1' then return ('1');
                        else return('0');
                        end if;
                end;
```

```
        function CONVERT_TO_BIT(X: X01Z)
        return BIT is
                begin
                        if X = '1' then return ('1');
                        else return('0');
                        end if;
                end;
end CONVERT_TYPE;

-- Library BTYPES contains a type declaration for the STATES type
package BTYPES is
        type STATES is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10);
end BTYPES;

library IEEE;
use work.BTYPES.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
-- Library FIND_PARAMETERS contains 7 functions:
-- FIND_M returns M = log2(N); width of address bus
-- FIND_S returns S = log2(M); width of SUMZERO bus
-- FIND_NUMXOR returns NUMXOR, the number of pairs of address lines
-- FIND_NUMDBGR returns the number of backgrounds for V=2 and 3
--                              returns 1/2 number of backgrounds for V=4
-- FIND_NUMDBGR1 returns the number of backgrounds in base matrix
-- FIND_NUMDBGR2 returns the number of backgrounds in Zones I, II base matrix
-- FIND_LFSRDEG returns degree of LFSR, if log2(N+1)+2 <= 16, return 16
--                                      if 16 < log2(N+1)+2 <= 20, return 20
--                                      if 20 < log2(N+1)+2 <= 25, return 25
--                                      if 25 < log2(N+1)+2 <= 30, return 30

package FIND_PARAMETERS is
        function FIND_M(N:INTEGER range 4 to 67108864)
        return INTEGER;
        function FIND_S(M:INTEGER)
        return INTEGER;
        function FIND_NUMXOR(M:INTEGER;V:INTEGER range 2 to 25)
        return INTEGER;
        function FIND_NUMDBGR(V:INTEGER range 2 to 25;M,S,NUMXOR: INTEGER)
        return INTEGER;
        function FIND_NUMDBGR1(M,NUMXOR: INTEGER)
        return INTEGER;
        function FIND_NUMDBGR2(M,NUMXOR: INTEGER)
        return INTEGER;
        function FIND_LFSRDEG(N:INTEGER range 4 to 67108864)
        return INTEGER;
end FIND_PARAMETERS;

package body FIND_PARAMETERS is
        function FIND_M(N:INTEGER range 4 to 67108864)
        return INTEGER is
                variable I:INTEGER;
                variable M:INTEGER;
                begin
                        if N > 2 and N <= 4 then M := 2;
                        elsif N > 4 and N <= 8 then M := 3;
                        elsif  N > 8 and N <= 16 then M := 4;
```

```
                elsif N > 16 and N <= 32 then M := 5;
                elsif N > 32 and N <= 64 then M := 6;
                elsif N > 64 and N <= 128 then M := 7;
                elsif N > 128 and N <= 256 then M := 8;
                elsif N > 256 and N <= 512 then M := 9;
                elsif N > 512 and N <= 1024 then M := 10;
                elsif N > 1024 and N <= 2048 then M := 11;
                elsif N > 2048 and N <= 4096 then M := 12;
                elsif N > 4096 and N <= 8192 then M := 13;
                elsif N > 8192 and N <= 16384 then M := 14;
                elsif N > 16384 and N <= 32768 then M := 15;
                elsif N > 32768 and N <= 65536 then M := 16;
                elsif N > 65536 and N <= 131072 then M := 17;
                elsif N > 131072 and N <= 262144 then M := 18;
                elsif N > 262144 and N <= 524288 then M := 19;
                elsif N > 524288 and N <= 1048576 then M := 20;
                elsif N > 1048576 and N <= 2097152 then M := 21;
                elsif N > 2097152 and N <= 4194304 then M := 22;
                elsif N > 4194304 and N <= 8388608 then M := 23;
                elsif N > 8388608 and N <= 16777216 then M := 24;
                elsif N > 16777216 and N <= 33554432 then M := 25;
                elsif N > 33554432 then M := 26;
                end if;
                return M;
        end;
function FIND_S(M:INTEGER)
return INTEGER is
        variable S:INTEGER;
        begin
                if M >=2 and M < 4 then S := 2;
                elsif M >= 4 and M < 8 then S := 3;
                elsif M >= 8 and M < 16 then S := 4;
                elsif M >= 16  then S := 5;
                end if;
                return S;
        end;
function FIND_NUMXOR(M:INTEGER;V:INTEGER range 2 to 25)
return INTEGER is
        variable NUMXOR: INTEGER;
        variable X : INTEGER;
        begin
                if V >= 15 then
                        X := M/2;
                else
                        X := M;
                end if;
                NUMXOR := X*(X-1)/2;
                return NUMXOR;
        end ;
```

```
        function FIND_NUMDBGR(V:INTEGER range 2 to 25;M,S,NUMXOR:INTEGER)
        return INTEGER is
                variable NUMDBGR:INTEGER;
                begin
                        if V = 2 then
                                NUMDBGR := 2;
                        elsif V = 3 then
                                NUMDBGR := 2 + M + S;
                        else
                                NUMDBGR := 1 + M + NUMXOR;
                        end if;
                        return NUMDBGR;
                end;
        function FIND_NUMDBGR1(M,NUMXOR:INTEGER)
        return INTEGER is
                variable NUMDBGR1:INTEGER;
                begin
                        NUMDBGR1 := M/2 + NUMXOR;
                        return NUMDBGR1;
                end;
        function FIND_NUMDBGR2(M,NUMXOR:INTEGER)
        return INTEGER is
                variable NUMDBGR2:INTEGER;
                begin
                        NUMDBGR2 := M/2;
                        return NUMDBGR2;
                end;
        function FIND_LFSRDEG(N:INTEGER range 4 to 67108864)
        return INTEGER is
                variable LFSRDEG:INTEGER;
                begin
                        LFSRDEG := FIND_M(N+1)+2;
                        if LFSRDEG <= 16 then
                                return 16;
                        elsif LFSRDEG <= 20 then
                                return 20;
                        elsif LFSRDEG <=25 then
                                return 25;
                        else
                                return 30;
                        end if;
                end;
end FIND_PARAMETERS;
```

BIST Circuit

File: bist.vhd

```
-- This file contains the structural description of the BIST circuit

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.CONVERT_TYPE.all;   -- include functions from library CONVERT_TYPE
use work.BTYPES.all;         -- include type declarations from library CONVERT_TYPE
use work.FIND_PARAMETERS.all;-- include functions from library FIND_PARAMETERS

entity BIST is
generic (N: INTEGER range 4 to 67108864; V:INTEGER range 2 to 25);
-- The input parameters:
-- N : Size of (N x 1) memory to be tested range from 8 bit to 64Mbit
-- V : Type of faults;      V=2   for 2-coupling fault
--                          V=3   for 3-coupling fault
--                          V=4   for 4-coupling fault
--                          V=15  for scrambled 5-cell NPSF with T neighborhoods
--                          V=25  for scrambled 5-cell NPSF with neighborhoods of Type 1

-- The input and output declarations:
port(   CLK             : in BIT;                           --system clock
        CLR             : in BIT;                           --system reset
        START_BIST      : in BIT;
        BISTACK         : in BIT;
        BIST_BUSY       : out BIT;
        BIST_DONE       : out BIT;
        READ            : out BIT;                          --BIST controller READ
        WRITE           : out BIT;                          --BIST controller WRITE
        ADDR            : out INTEGER range 0 to N-1;       --BIST controller address
        PASSFAIL        : out BIT;                          --PASS/FAIL signal
        DATA            : inout X01Z);                      --bidirectional data bus
end BIST;

architecture STRUCTURE of BIST is
        CONSTANT M:INTEGER              := FIND_M(N);
        CONSTANT S:INTEGER              := FIND_S(M);
        CONSTANT NUMXOR :INTEGER        := FIND_NUMXOR(M,V);
        CONSTANT NUMDBGR:INTEGER        := FIND_NUMDBGR(V,M,S,NUMXOR);
        CONSTANT NUMDBGR1:INTEGER       := FIND_NUMDBGR1(M,NUMXOR);
        CONSTANT NUMDBGR2:INTEGER       := FIND_NUMDBGR2(M,NUMXOR);
        CONSTANT LFSRDEG :INTEGER       := FIND_LFSRDEG(N);
--M = log2(N), width of address bus
--S = log2(M), number of bits to represent the number of zeros in address bus
--NUMXOR     = M choose 2, the number of address pairs for V=4, M/2 choose 2 for V=5t, 5x
--NUMDBGR    = number of backgrounds for V=2,3
--          = 1/2 the number of backgrounds for V=4
--NUMDBGR1   =number of backgrounds in the base matrix(V=5t and 5x)
--NUMDBGR2   =number of backgrounds in Zones I and II of the base matrix(V=5t)
--LFSRDEG    =log2(N+1)+2, degree of LFSR

-- Internal signal declarations

        signal ADDRA            : INTEGER range 0 to N-1;       --unlatched address
        signal ALAST_ADDR       : BIT;                          --unlatched last address
```

```
        signal LAST_ADDR      : BIT;                                    --latched last address
        signal DBGR           : INTEGER range 0 to NUMDBGR-1;--background number for V=2,3,4
        signal DBGRC          : BIT;                                    --DBGRC  for V=4
        signal ADBGR          : INTEGER range 0 to NUMDBGR-1;           --ADBGR  for V=4
        signal ADBGRC         : BIT;                                    --ADBGRC  for V=4
        signal TDBGR1         : INTEGER range 0 to NUMDBGR1;            --DBGR1  for V=5t
        signal TDBGR2         : INTEGER range 0 to NUMDBGR2;            --DBGR2  for V=5t
        signal TDBGRT         : BIT;                                    --DBGRT  for V=5t
        signal TDBGRC         : BIT;                                    --DBGRC  for V=5t
        signal TADBGR1        : INTEGER range 0 to NUMDBGR1;            --ADBGR1 for V=5t
        signal TADBGR2        : INTEGER range 0 to NUMDBGR2;            --ADBGR1 for V=5t
        signal TADBGRT        : BIT;                                    --ADBGRT for V=5t
        signal TADBGRC        : BIT;                                    --ADBGRC for V=5t
        signal XDBGR1         : INTEGER range 0 to NUMDBGR1;            --DBGR1  for V=5x
        signal XDBGR2         : INTEGER range 0 to NUMDBGR1;            --DBGR2  for V=5x
        signal XDBGRC         : BIT;                                    --DBGRC  for V=5x
        signal XADBGR1        : INTEGER range 0 to NUMDBGR1;            --ADBGR1 for V=5x
        signal XADBGR2        : INTEGER range 0 to NUMDBGR1;            --ADBGR2 for V=5x
        signal XADBGRC        : BIT;                                    --ADBGRC for V=5x
        signal LAST_DBGR      : BIT;                                    --last background
        signal APEQN          : BIT;                                    --unlatched P=N signal
        signal PEQN           : BIT;                                    --latched P=N signal
        signal NEXT_ADDR      : BIT;                                    --increment address signal
        signal NEXT_DBGR      : BIT;                                    --increment background signal
        signal LOADREG1       : BIT;                                    --load register 1 signal
        signal LOADREG2       : BIT;                                    --load register 2 signal
        signal CLRSIG         : BIT;                                    --clear LFSR signal
        signal CRESET         : BIT;                                    --BIST controller reset
        signal STATE          : STATES;                                 --state variable
        signal ASYNCSTATE     : STATES;                                 --next state variable
        signal DB             : X01Z;                                   --internal bidirectional data bus
        signal IREAD          : BIT;                                    --internal BIST controller read
        signal IWRITE         : BIT;                                    --internal BIST controller write
        signal PHASE2         : BIT;                                    --PHASE2 signal
        signal LPHASE2        : BIT;                                    --set PHASE2 signal
        signal CLRPHASE2      : BIT;                                    --clear PHASE2 signal
        signal PASS           : BIT;                                    --PASS signal
        signal PF             : BIT;                                    --internal PASS/FAIL signal
        signal FAIL           : BIT;                                    --FAIL signal

-- Components Declarations:

-- BIST CONTROLLER

component BIST_CONTROLLER
    port(CLK          : in    BIT;
         RESET        : in    BIT;
         START_BIST   : in    BIT;
         BISTACK      : in    BIT;
         LAST_ADDR    : in    BIT;
         ALAST_ADDR   : in    BIT;
         LAST_DBGR    : in    BIT;
         PEQN         : in    BIT;
         APEQN        : in    BIT;
         PHASE2       : in    BIT;
         PASS         : in    BIT;
         LPHASE2      : out   BIT;
         CLRPHASE2    : out   BIT;
```

```
        READ            : out   BIT;
        WRITE           : out   BIT;
        BIST_BUSY       : out   BIT;
        BIST_DONE       : out   BIT;
        NEXT_ADDR       : out   BIT;
        NEXT_DBGR       : out   BIT;
        LOADREG1        : out   BIT;
        LOADREG2        : out   BIT;
        CLRSIG          : out   BIT;
        CRESET          : out   BIT;
        FAIL            : out   BIT;
        STATE           : out   STATES;
        ASYNCSTATE      : out   STATES);
end component;

-- DATA PATH

-- Address Generator
component ADDRESS_GENERATOR
   generic(N:INTEGER);
   port(CLK             : in    Bit;
        RESET           : in    Bit;
        NEXT_ADDR       : in    Bit;
        ALAST_ADDR      : out   Bit;
        LAST_ADDR       : out   Bit;
        ADDR            : out   INTEGER range 0 to N-1;
        ADDRA           : out   INTEGER range 0 to N-1);
end component;

-- Background Counter V=2,3
component BACKGROUND_COUNTER
   generic(NUMDBGR:INTEGER);
   port(CLK             : in    BIT;
        RESET           : in    BIT;
        NEXT_DBGR       : in    BIT;
        LAST_DBGR       : out   BIT;
        DBGR            : out   INTEGER range 0 to NUMDBGR-1);
end component;

-- Background Counter V=4
component BACKGROUND_COUNTER_V4
   generic(NUMDBGR:INTEGER);
   port(CLK             : in    BIT;
        RESET           : in    BIT;
        NEXT_DBGR       : in    BIT;
        LAST_DBGR       : out   BIT;
        DBGR            : out   INTEGER range 0 to NUMDBGR-1;
        DBGRC           : out   BIT;
        ADBGR           : out   INTEGER range 0 to NUMDBGR-1;
        ADBGRC          : out   BIT);
end component;
```

```
-- Background Counter V=5t
component BACKGROUND_COUNTER_V5T
    generic(NUMDBGR1:INTEGER;NUMDBGR2:INTEGER);
    port(CLK            : in    BIT;
         RESET          : in    BIT;
         NEXT_DBGR      : in    BIT;
         LAST_DBGR      : out   BIT;
         DBGR1          : out   INTEGER range 0 to NUMDBGR1;
         DBGR2          : out   INTEGER range 0 to NUMDBGR2;
         DBGRT          : out   BIT;
         DBGRC          : out   BIT;
         ADBGR1         : out   INTEGER range 0 to NUMDBGR1;
         ADBGR2         : out   INTEGER range 0 to NUMDBGR2;
         ADBGRT         : out   BIT;
         ADBGRC         : out   BIT);
end component;
-- Background Counter V=5x
component BACKGROUND_COUNTER_V5X
    generic(NUMDBGR1:INTEGER);
    port(CLK            : in    BIT;
         RESET          : in    BIT;
         NEXT_DBGR      : in    BIT;
         LAST_DBGR      : out   BIT;
         DBGR1          : out   INTEGER range 0 to NUMDBGR1;
         DBGR2          : out   INTEGER range 0 to NUMDBGR1;
         DBGRC          : out   BIT;
         ADBGR1         : out   INTEGER range 0 to NUMDBGR1;
         ADBGR2         : out   INTEGER range 0 to NUMDBGR1;
         ADBGRC         : out   BIT);
end component;

-- Background Logic V=2
component BACKGROUND_LOGIC_V2
    port(APEQN          : out   BIT;
         PEQN           : out   BIT);
end component;

-- Background Logic V=3
component BACKGROUND_LOGIC_V3
    generic(N:INTEGER;M:INTEGER;NUMDBGR:INTEGER;S:INTEGER);
    port(CLK            : in    BIT;
         RESET          : in    BIT;
         ADDRESS        : in    INTEGER range 0 to N-1;
         DBGR           : in    INTEGER range 0 to NUMDBGR-1;
         NEXT_ADDR      : in    BIT;
         APEQN          : out   BIT;
         PEQN           : out   BIT);
end component;
```

```
-- Background Logic V=4
component BACKGROUND_LOGIC_V4
   generic(NUMXOR:INTEGER;N:INTEGER;M:INTEGER;NUMDBGR:INTEGER);
   port(CLK          : in    BIT;
        RESET        : in    BIT;
        ADDRESS      : in    INTEGER range 0 to N-1;
        DBGR         : in    INTEGER range 0 to NUMDBGR-1;
        DBGRC        : in    BIT;
        ADBGR        : in    INTEGER range 0 to NUMDBGR-1;
        ADBGRC       : in    BIT;
        NEXT_ADDR    : in    BIT;
        PEQN         : out   BIT;
        APEQN        : out   BIT);
end component;

-- Background Logic V=5t
component BACKGROUND_LOGIC_V5T
   generic(N:INTEGER;M:INTEGER;NUMXOR:INTEGER;NUMDBGR1:INTEGER;
           NUMDBGR2:INTEGER);
   port(CLK          : in    BIT;
        RESET        : in    BIT;
        ADDRESS      : in    INTEGER range 0 to N-1;
        DBGR1        : in    INTEGER range 0 to NUMDBGR1;
        DBGR2        : in    INTEGER range 0 to NUMDBGR2;
        DBGRT        : in    BIT;
        DBGRC        : in    BIT;
        ADBGR1       : in    INTEGER range 0 to NUMDBGR1;
        ADBGR2       : in    INTEGER range 0 to NUMDBGR2;
        ADBGRT       : in    BIT;
        ADBGRC       : in    BIT;
        NEXT_ADDR    : in    BIT;
        APEQN        : out   BIT;
        PEQN         : out   BIT);
end component;

-- Background Logic V=5x
component BACKGROUND_LOGIC_V5X
   generic(N:INTEGER;M:INTEGER;NUMXOR:INTEGER;NUMDBGR1:INTEGER);
   port(CLK          : in    BIT;
        RESET        : in    BIT;
        ADDRESS      : in    INTEGER range 0 to N-1;
        DBGR1        : in    INTEGER range 0 to NUMDBGR1;
        DBGR2        : in    INTEGER range 0 to NUMDBGR1;
        DBGRC        : in    BIT;
        ADBGR1       : in    INTEGER range 0 to NUMDBGR1;
        ADBGR2       : in    INTEGER range 0 to NUMDBGR1;
        ADBGRC       : in    BIT;
        NEXT_ADDR    : in    BIT;
        APEQN        : out   BIT;
        PEQN         : out   BIT);
end component;
```

```
-- Response Analyzer
component RESPONSE_ANALYZER
    generic(LFSRDEG:INTEGER);
    port(CLK       : in     BIT;
         RESET     : in     BIT;
         CLRSIG    : in     BIT;
         ENABLE    : in     BIT;
         LOADREG1  : in     BIT;
         LOADREG2  : in     BIT;
         B         : in     BIT;
         PEQN      : in     BIT;
         PHASE2    : in     BIT;
         STATE     : in     STATES;
         ASYNCSTATE : in    STATES;
         PASS      : out    BIT);

end component;

-- Test Pattern Generator
component TEST_PATTERN_GENERATOR
    port(CLK       : in     BIT;
         RESET     : in     BIT;
         WRITE     : in     BIT;
         READ      : in     BIT;
         B         : in     BIT;
         Q         : out    X01Z);
end component;

-- Phase2 Latch
component PHASE2LATCH
    port(CLR       : in     BIT;
         CLRPHASE2 : in     BIT;
         EN        : in     BIT;
         D         : in     BIT;
         Q         : out    BIT);
end component;

-- PASS/FAIL Latch
component PASSLATCH
    port(CLR       : in     BIT;
         EN        : in     BIT;
         D         : in     BIT;
         Q         : out    BIT);
end component;

begin
         PASSFAIL  <= PF;
         READ      <= IREAD;
         WRITE     <= IWRITE;
         DATA      <= DB;
```

```
BIST_CONTROL: BIST_CONTROLLER
port map(CLK          =>    CLK,
         RESET        =>    CLR,
         START_BIST   =>    START_BIST,
         BISTACK      =>    BISTACK,
         LAST_ADDR    =>    LAST_ADDR,
         ALAST_ADDR   =>    ALAST_ADDR,
         LAST_DBGR    =>    LAST_DBGR,
         PEQN         =>    PEQN,
         APEQN        =>    APEQN,
         PHASE2       =>    PHASE2,
         PASS         =>    PASS,
         LPHASE2      =>    LPHASE2,
         CLRPHASE2    =>    CLRPHASE2,
         READ         =>    IREAD,
         WRITE        =>    IWRITE,
         BIST_BUSY    =>    BIST_BUSY,
         BIST_DONE    =>    BIST_DONE,
         NEXT_ADDR    =>    NEXT_ADDR,
         NEXT_DBGR    =>    NEXT_DBGR,
         LOADREG1     =>    LOADREG1,
         LOADREG2     =>    LOADREG2,
         CLRSIG       =>    CLRSIG,
         CRESET       =>    CRESET,
         FAIL         =>    FAIL,
         STATE        =>    STATE,
         ASYNCSTATE   =>    ASYNCSTATE);

ADDR_GEN: ADDRESS_GENERATOR
generic map(N=>N)
port map(CLK          =>    CLK,
         RESET        =>    CRESET,
         NEXT_ADDR    =>    NEXT_ADDR,
         ALAST_ADDR   =>    ALAST_ADDR,
         LAST_ADDR    =>    LAST_ADDR,
         ADDR         =>    ADDR,
         ADDRA        =>    ADDRA);

-- Selectively generate background counter based of the value of V
    SELECT_BGC:
    if V <= 3 generate
    BG_COUNT: BACKGROUND_COUNTER
    generic map(NUMDBGR=>NUMDBGR)
    port map(CLK          =>    CLK,
             RESET        =>    CRESET,
             NEXT_DBGR    =>    NEXT_DBGR,
             LAST_DBGR    =>    LAST_DBGR,
             DBGR         =>    DBGR);
    end generate;
```

```
SELECT_BGCV4:
if V = 4 generate
BG_COUNT_V4: BACKGROUND_COUNTER_V4
generic map(NUMDBGR=>NUMDBGR)
port map(CLK            =>      CLK,
         RESET          =>      CRESET,
         NEXT_DBGR      =>      NEXT_DBGR,
         LAST_DBGR      =>      LAST_DBGR,
         DBGR           =>      DBGR,
         DBGRC          =>      DBGRC,
         ADBGR          =>      ADBGR,
         ADBGRC         =>      ADBGRC);
end generate;

SELECT_BGCV5T:
if V = 15 generate
BG_COUNT_V5T: BACKGROUND_COUNTER_V5T
generic map(NUMDBGR1=>NUMDBGR1,NUMDBGR2=>NUMDBGR2)
port map(CLK            =>      CLK,
         RESET          =>      CRESET,
         NEXT_DBGR      =>      NEXT_DBGR,
         LAST_DBGR      =>      LAST_DBGR,
         DBGR1          =>      TDBGR1,
         DBGR2          =>      TDBGR2,
         DBGRT          =>      TDBGRT,
         DBGRC          =>      TDBGRC,
         ADBGR1         =>      TADBGR1,
         ADBGR2         =>      TADBGR2,
         ADBGRT         =>      TADBGRT,
         ADBGRC         =>      TADBGRC);
end generate;

SELECT_BGCV5X:
if V = 25 generate
BG_COUNT_V5X: BACKGROUND_COUNTER_V5X
generic map(NUMDBGR1=>NUMDBGR1)
port map(CLK            =>      CLK,
         RESET          =>      CRESET,
         NEXT_DBGR      =>      NEXT_DBGR,
         LAST_DBGR      =>      LAST_DBGR,
         DBGR1          =>      XDBGR1,
         DBGR2          =>      XDBGR2,
         DBGRC          =>      XDBGRC,
         ADBGR1         =>      XADBGR1,
         ADBGR2         =>      XADBGR2,
         ADBGRC         =>      XADBGRC);
end generate;

-- Selectively generate background logic based of the value of V
SELECT_BG2:
if V = 2 generate
BGLOGIC_V2: BACKGROUND_LOGIC_V2
port map(APEQN          =>      APEQN,
         PEQN           =>      PEQN );
end generate;
```

```
SELECT_BG3:
if V = 3 generate
BGLOGIC_V3: BACKGROUND_LOGIC_V3
generic map(N=>N, M=>M,NUMDBGR=>NUMDBGR,S=>S)
port map(CLK          =>    CLK,
         RESET        =>    CRESET,
         ADDRESS      =>    ADDRA,
         DBGR         =>    DBGR ,
         NEXT_ADDR    =>    NEXT_ADDR,
         APEQN        =>    APEQN,
         PEQN         =>    PEQN);
end generate;

SELECT_BG4:
if V = 4 generate
BGLOGIC_V4: BACKGROUND_LOGIC_V4
generic map(NUMXOR=>NUMXOR,N=>N,M=>M,NUMDBGR=>NUMDBGR)
port map(CLK          =>    CLK,
         RESET        =>    CRESET,
         ADDRESS      =>    ADDRA,
         DBGR         =>    DBGR,
         DBGRC        =>    DBGRC,
         ADBGR        =>    ADBGR,
         ADBGRC       =>    ADBGRC,
         NEXT_ADDR    =>    NEXT_ADDR,
         PEQN         =>    PEQN,
         APEQN        =>    APEQN);
end generate;

SELECT_BG5T:
if V = 15 generate
BGLOGIC_V5T: BACKGROUND_LOGIC_V5T
generic map(N=>N,M=>M,NUMXOR=>NUMXOR,NUMDBGR1=>NUMDBGR1,
NUMDBGR2=>NUMDBGR2)
port map(CLK          =>    CLK,
         RESET        =>    CRESET,
         ADDRESS      =>    ADDRA,
         DBGR1        =>    TDBGR1,
         DBGR2        =>    TDBGR2,
         DBGRT        =>    TDBGRT,
         DBGRC        =>    TDBGRC,
         ADBGR1       =>    TADBGR1,
         ADBGR2       =>    TADBGR2,
         ADBGRT       =>    TADBGRT,
         ADBGRC       =>    TADBGRC,
         NEXT_ADDR    =>    NEXT_ADDR,
         APEQN        =>    APEQN,
         PEQN         =>    PEQN);
end generate;
```

```
        SELECT_BG5X:
        if V = 25 generate
        BGLOGIC_V5X: BACKGROUND_LOGIC_V5X
        generic map(N =>N,M=>M,NUMXOR=>NUMXOR,NUMDBGR1=>NUMDBGR1)
        port map(CLK            =>      CLK,
                 RESET          =>      CRESET,
                 ADDRESS        =>      ADDRA,
                 DBGR1          =>      XDBGR1,
                 DBGR2          =>      XDBGR2,
                 DBGRC          =>      XDBGRC,
                 ADBGR1         =>      XADBGR1,
                 ADBGR2         =>      XADBGR2,
                 ADBGRC         =>      XADBGRC,
                 NEXT_ADDR      =>      NEXT_ADDR,
                 APEQN          =>      APEQN,
                 PEQN           =>      PEQN);
        end generate;

        RESP_ALYZR: RESPONSE_ANALYZER
        generic map(LFSRDEG=>LFSRDEG)
        port map(CLK            =>      CLK,
                 RESET          =>      CRESET,
                 CLRSIG         =>      CLRSIG,
                 ENABLE         =>      IREAD,
                 LOADREG1       =>      LOADREG1,
                 LOADREG2       =>      LOADREG2,
                 B              =>      CONVERT_TO_BIT(DB),
                 PEQN           =>      PEQN,
                 PHASE2         =>      PHASE2,
                 STATE          =>      STATE,
                 ASYNCSTATE     =>      ASYNCSTATE,
                 PASS           =>      PASS);

        TPG: TEST_PATTERN_GENERATOR
        port map(CLK            =>      CLK,
                 RESET          =>      CRESET,
                 WRITE          =>      IWRITE,
                 READ           =>      IREAD,
                 B              =>      CONVERT_TO_BIT(DB),
                 Q              =>      DB);

        P2FF:PHASE2LATCH
        port map(CLR            =>      CRESET,
                 CLRPHASE2      =>      CLRPHASE2,
                 EN             =>      LPHASE2,
                 D              =>      PHASE2,
                 Q              =>      PHASE2);

        PFF: PASSLATCH
        port map(CLR            =>      FAIL,
                 EN             =>      CRESET,
                 D              =>      PF,
                 Q              =>      PF);
end STRUCTURE;
```

BIST_Controller

File: bist_cont.vhd

```
use work.BTYPES.all;

entity BIST_CONTROLLER is
  port(CLK              : in  BIT;        --system clock
       RESET            : in  BIT;        --system reset
       START_BIST       : in  BIT;        --start BIST signal
       BISTACK          : in  BIT;        --exit BIST signal
       LAST_ADDR        : in  BIT;        --latched last address signal
       ALAST_ADDR       : in  BIT;        --unlatched last address signal
       LAST_DBGR        : in  BIT;        --latched last databackground signal
       PEQN             : in  BIT;        --latched P bit equal N bit signal
       APEQN            : in  BIT;        --unlatched P bit equal N bit signal
       PHASE2           : in  BIT;        --phase2 signal
       PASS             : in  BIT;        --Pass input from response analyzer
       LPHASE2          : out BIT;        --set PHASE2 latch
       CLRPHASE2        : out BIT;        --clear PHASE2 latch
       READ             : out BIT;        --BIST controller read output
       WRITE            : out BIT;        --BIST controller write output
       BIST_BUSY        : out BIT;        --BIST_BUSY output
       BIST_DONE        : out BIT;        --BIST_DONE output
       NEXT_ADDR        : out BIT;        --increment address counter output
       NEXT_DBGR        : out BIT;        --increment background counter output
       LOADREG1         : out BIT;        --load register signal
       LOADREG2         : out BIT;        --load register signal
       CLRSIG           : out BIT;        --clear signature register signal
       CRESET           : out BIT;        --controller reset signal
       FAIL             : out BIT;        --FAIL output to PASS/FAIL latch
       STATE            : out STATES;     --current state of type STATES
       ASYNCSTATE : out STATES);          --next state
end BIST_CONTROLLER;

architecture BEHAVIORAL_DESCRIPTION of BIST_CONTROLLER is
       signal ILPHASE2          : BIT;          --unlatched set PHASE2 signal
       signal ICLRPHASE2        : BIT;          --unlatched clear PHASE2 signal
       signal IREAD             : BIT;          --unlatched read signal
       signal IWRITE            : BIT;          --unlatched write signal
       signal IBIST_BUSY        : BIT;          --unlatched BIST_BUSY signal
       signal IBIST_DONE        : BIT;          --unlatched BIST_DONE signal
       signal INEXT_ADDR        : BIT;          --unlatched increment address counter signal
       signal INEXT_DBGR        : BIT;          --unlatched increment background counter signal
       signal ILOADREG1         : BIT;          --unlatched load register 1 signal
       signal ILOADREG2         : BIT;          --unlatched load register 2 signal
       signal ICLRSIG           : BIT;          --unlatched clear signature register signal
       signal ICRESET           : BIT;          --unlatched BIST controller reset
       signal IFAIL             : BIT;          --unlatched FAIL signal
       signal ISTATE            : STATES;       --internal current state variable
       signal ASTATE            : STATES;       --internal next state variable
begin
       STATE                    <= ISTATE;
       ASYNCSTATE               <= ASTATE;
       P_FSM : process(ISTATE,START_BIST,BISTACK,ALAST_ADDR,LAST_ADDR,LAST_DBGR,
               PEQN,APEQN,PHASE2)
       begin
```

```
case ISTATE is
        when S0  =>    if START_BIST = '1' then  --S0 branch condition
                               ASTATE <= S1;              --assign next state to S1
                       else
                               ASTATE <= S0;              --assign next state to S0
                       end if;
        when S1  =>    if PHASE2 = '1' then
                               ASTATE <= S2;
                       else
                               ASTATE <= S3;
                       end if;
        when S2  =>    ASTATE <= S3;
        when S3  =>    if PHASE2 = '0' and LAST_ADDR = '1' then
                               ASTATE <= S5;
                       elsif PHASE2 = '0' and LAST_ADDR = '0' then
                               ASTATE <= S1;
                       else
                               ASTATE <= S4;
                       end if;
        when S4  =>    if LAST_ADDR = '1'  then
                               ASTATE <= S5;
                       else
                               ASTATE <= S1;
                       end if;
        when S5  =>    if  (PHASE2 = '0' and LAST_ADDR = '1') or
                       (PEQN = '1' and LAST_ADDR ='1') then
                               ASTATE <= S7;
                       elsif (PHASE2 = '0' and LAST_ADDR = '0') or
                       (PEQN = '1' and LAST_ADDR = '0') then
                               ASTATE <= S5;
                       else
                               ASTATE <= S6;
                       end if;
        when S6  =>    if LAST_ADDR = '0' then
                               ASTATE <= S5;
                       else
                               ASTATE <= S7;
                       end if;
        when S7  =>    if LAST_ADDR = '1' and PHASE2 = '1' then
                               ASTATE <= S8;
                       elsif LAST_ADDR = '1' and PHASE2 = '0' then
                               ASTATE <= S9;
                       else
                               ASTATE <= S7;
                       end if;
        when S8  =>    if LAST_DBGR = '1' then
                               ASTATE <= S10;
                       else
                               ASTATE <= S9;
                       end if;
        when S9  =>    ASTATE <= S1;
        when S10 =>    if BISTACK = '1' then
                               ASTATE <= S0;
                       else
                               ASTATE <= S10;
                       end if;
    end case;
end process P_FSM;
```

```
P_OUTPUT: process (ASTATE,PHASE2,PASS,ALAST_ADDR,APEQN,LAST_ADDR)
begin
-- assign unlatched controller output values in the next state
        case ASTATE is
                when S0  =>    ILPHASE2      <= '0';
                               ICLRPHASE2    <= '0';
                               IREAD         <= '0';
                               IWRITE        <= '0';
                               IBIST_BUSY    <= '0';
                               IBIST_DONE    <= '0';
                               INEXT_ADDR    <= '0';
                               INEXT_DBGR    <= '0';
                               ILOADREG1     <= '0';
                               ILOADREG2     <= '0';
                               ICLRSIG       <= '0';
                               ICRESET       <= '0';
                               IFAIL         <= '0';
                when S1  =>    ILPHASE2      <= '0';
                               ICLRPHASE2    <= '0';
                               IREAD         <= '1';
                               IWRITE        <= '0';
                               IBIST_BUSY    <= '1';
                               IBIST_DONE    <= '0';
                               INEXT_ADDR    <= '0';
                               INEXT_DBGR    <= '0';
                               ILOADREG1     <= '0';
                               ILOADREG2     <= '0';
                               ICLRSIG       <= '0';
                               ICRESET       <= '1';
                               IFAIL         <= '0';
                when S2  =>    ILPHASE2      <= '0';
                               ICLRPHASE2    <= '0';
                               IREAD         <= '0';
                               IWRITE        <= '1';
                               IBIST_BUSY    <= '1';
                               IBIST_DONE    <= '0';
                               INEXT_ADDR    <= '0';
                               INEXT_DBGR    <= '0';
                               ILOADREG1     <= '0';
                               ILOADREG2     <= '0';
                               ICLRSIG       <= '0';
                               ICRESET       <= '1';
                               IFAIL         <= '0';
                when S3  =>    ILPHASE2      <= '0';
                               ICLRPHASE2    <= '0';
                               IREAD         <= '1';
                               IWRITE        <= '0';
                               IBIST_BUSY    <= '1';
                               IBIST_DONE    <= '0';
                               INEXT_ADDR    <= not(PHASE2);
                               INEXT_DBGR    <= '0';
                               ILOADREG1     <= not(PHASE2) and LAST_ADDR;
                               ILOADREG2     <= '0';
                               ICLRSIG       <= '0';
                               ICRESET       <= '1';
                               IFAIL         <= '0';
```

```
when S4  =>   ILPHASE2     <= '0';
              ICLRPHASE2   <= '0';
              IREAD        <= '0';
              IWRITE       <= '1';
              IBIST_BUSY   <= '1';
              IBIST_DONE   <= '0';
              INEXT_ADDR   <= '1';
              INEXT_DBGR   <= '0';
              ILOADREG1    <= '0';
              ILOADREG2    <= '0';
              ICLRSIG      <= '0';
              ICRESET      <= '1';
              IFAIL        <= not(PASS) and LAST_ADDR;

when S5  =>   ILPHASE2     <= '0';
              ICLRPHASE2   <= '0';
              IREAD        <= not(APEQN);
              IWRITE       <= '0';
              IBIST_BUSY   <= '1';
              IBIST_DONE   <= '0';
              INEXT_ADDR   <= not(PHASE2) or APEQN;
              INEXT_DBGR   <= '0';
              ILOADREG1    <= '0';
              ILOADREG2    <= '0';
              ICLRSIG      <= '0';
              ICRESET      <= '1';
              IFAIL        <= '0';

when S6  =>   ILPHASE2     <= '0';
              ICLRPHASE2   <= '0';
              IREAD        <= '0';
              IWRITE       <= '1';
              IBIST_BUSY   <= '1';
              IBIST_DONE   <= '0';
              INEXT_ADDR   <= '1';
              INEXT_DBGR   <= '0';
              ILOADREG1    <= '0';
              ILOADREG2    <= '0';
              ICLRSIG      <= '0';
              ICRESET      <= '1';
              IFAIL        <= '0';

when S7  =>   ILPHASE2     <= '0';
              ICLRPHASE2   <= '0';
              IREAD        <= '1';
              IWRITE       <= '0';
              IBIST_BUSY   <= '1';
              IBIST_DONE   <= '0';
              INEXT_ADDR   <= '1';
              INEXT_DBGR   <= '0';
              ILOADREG1    <= '0';
              ILOADREG2    <= not(PHASE2) and ALAST_ADDR;
              ICLRSIG      <= '0';
              ICRESET      <= '1';
              IFAIL        <= '0';
```

```
            when S8  =>    ILPHASE2     <= '0';
                           ICLRPHASE2   <= '0';
                           IREAD        <= '0';
                           IWRITE       <= '0';
                           IBIST_BUSY   <= '1';
                           IBIST_DONE   <= '0';
                           INEXT_ADDR   <= '0';
                           INEXT_DBGR   <= '1';
                           ILOADREG1    <= '0';
                           ILOADREG2    <= '0';
                           ICLRSIG      <= '0';
                           ICRESET      <= '1';
                           IFAIL        <= not(PASS);

            when S9  =>    ILPHASE2     <= not(PHASE2);
                           ICLRPHASE2   <= PHASE2;
                           IREAD        <= '0';
                           IWRITE       <= '0';
                           IBIST_BUSY   <= '1';
                           IBIST_DONE   <= '0';
                           INEXT_ADDR   <= '0';
                           INEXT_DBGR   <= '0';
                           ILOADREG1    <= '0';
                           ILOADREG2    <= '0';
                           ICLRSIG      <= '1';
                           ICRESET      <= '1';
                           IFAIL        <= '0';

            when S10 =>    ILPHASE2     <= '0';
                           ICLRPHASE2   <= '0';
                 I         READ         <= '0';
                           IWRITE       <= '0';
                           IBIST_BUSY   <= '0';
                           IBIST_DONE   <= '1';
                           INEXT_ADDR   <= '0';
                           INEXT_DBGR   <= '0';
                           ILOADREG1    <= '0';
                           ILOADREG2    <= '0';
                           ICLRSIG      <= '0';
                           ICRESET      <= '1';
                           IFAIL        <= '0';

        end case;
    end process P_OUTPUT;
```

```
P_SYNC:process(RESET,CLK,ASTATE,ILPHASE2,ICLRPHASE2,IREAD,IWRITE,IBIST_BUSY,
IBIST_DONE,INEXT_ADDR,INEXT_DBGR,ILOADREG1,ILOADREG2,ICLRSIG,ICRESET)
begin
        if RESET = '0' then                             --asynchronous reset
                ISTATE          <= S0;                  --reset current state to S0
                LPHASE2         <= '0';                 --reset all controller outputs to 0
                CLRPHASE2       <= '0';
                READ            <= '0';
                WRITE           <= '0';
                BIST_BUSY       <= '0';
                BIST_DONE       <= '0';
                NEXT_ADDR       <= '0';
                NEXT_DBGR       <= '0';
                LOADREG1        <= '0';
                LOADREG2        <= '0';
                CLRSIG          <= '0';
                CRESET          <= '0';
                FAIL            <= '0';
        elsif CLK = '0' and CLK'event then              --synchronize to negative clock edge
                ISTATE          <= ASTATE;      --latch next state as current state
                LPHASE2         <= ILPHASE2;    --latch all controller ouputs
                CLRPHASE2       <= ICLRPHASE2;
                READ            <= IREAD;
                WRITE           <= IWRITE;
                BIST_BUSY       <= IBIST_BUSY;
                BIST_DONE       <= IBIST_DONE;
                NEXT_ADDR       <= INEXT_ADDR;
                NEXT_DBGR       <= INEXT_DBGR;
                LOADREG1        <= ILOADREG1;
                LOADREG2        <= ILOADREG2;
                CLRSIG          <= ICLRSIG;
                CRESET          <= ICRESET;
                FAIL            <= IFAIL;
        end if;
    end process P_SYNC;
end BEHAVIORAL_DESCRIPTION;
```

Address_Generator

File: add_gen.vhd

```
entity ADDRESS_GENERATOR is
        generic(N:INTEGER);                                 --N: size of RAM
        port(CLK        : in Bit;                           --system clock
        RESET           : in Bit;                           --bist controller reset
        NEXT_ADDR       : in Bit;                           --increment address signal
        ALAST_ADDR      : out Bit;                          --unlatched last address
        LAST_ADDR       : out Bit;                          --latched last address signal
        ADDR            : out INTEGER range 0 to N-1;       --address signal (latched) counts
                                                            --from 0 to N-1
        ADDRA           : out INTEGER range 0 to N-1);      --address + 1 signal (unlatched)
end ADDRESS_GENERATOR;


architecture BEHAVIORAL_DESCRIPTION of ADDRESS_GENERATOR is
        signal iADDR    : INTEGER range 0 to N-1;           --internal ADDR signal
        signal iADDRA   : INTEGER range 0 to N-1;           --internal ADDRA signal
        signal iLAST_ADDR       : Bit;                      --internal LAST_ADDR signal
        signal iALAST_ADDR      : Bit;                      --internal ALAST_ADDR signal
begin
        ADDR<=iADDR;
        ADDRA<=iADDRA;
        iADDRA<=(iADDR+1) mod N;                            --address + 1
        ALAST_ADDR <= iALAST_ADDR;
        P_LASTADDR : process (iADDRA)
        begin
                if iADDRA=N-1 then
                        iALAST_ADDR <= '1';                 --ALAST_ADDR = 1 if ADDRA = N-1
                else
                        iALAST_ADDR <= '0';
                end if;
        end process P_LASTADDR;
        P_COUNT: process (CLK,RESET,iADDR,iADDRA,NEXT_ADDR,iALAST_ADDR)
        begin
                if RESET = '0' then
                        iADDR <= 0;
                        LAST_ADDR <= '0';
                elsif CLK='0' and CLK'EVENT then            --synchronize to negative clock edge
                        if NEXT_ADDR = '1' then
                                iADDR<=iADDRA;
                                LAST_ADDR <= iALAST_ADDR;
                        end if;
                end if;
        end process P_COUNT;
end BEHAVIORAL_DESCRIPTION;
```

Background Counter (V=2 and V=3)

File: background_counter.vhd

```
entity BACKGROUND_COUNTER is
        generic(NUMDBGR : INTEGER);             --NUMDBGR: number of backgrounds
        port(CLK        : in BIT;               --system clock
        RESET           : in BIT;               --bist controller reset
        NEXT_DBGR       : in BIT;               --increment background counter signal
        LAST_DBGR       : out BIT;              --last data background signal
        DBGR            : out INTEGER range 0 to NUMDBGR-1); --background number
end BACKGROUND_COUNTER;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_COUNTER is
        signal iDBGR : INTEGER range 0 to NUMDBGR-1;        --internal DBGR signal
begin
        DBGR<=iDBGR;
        P_LASTDBGR: process(iDBGR)
        begin
                if iDBGR = NUMDBGR - 1 then             --last data background
                        LAST_DBGR <= '1';
                else
                        LAST_DBGR <= '0';
                end if;
        end process P_LASTDBGR;
        D_COUNT: process (CLK,RESET,iDBGR,NEXT_DBGR)
        begin
                if RESET = '0' then                     --asynchronous reset
                        iDBGR <= 0;
                elsif CLK='0' and CLK'event then  --synchronize to negative clock edge
                        if NEXT_DBGR = '1' then
                                if iDBGR = NUMDBGR - 1 then
                                        iDBGR <= 0;
                                else
                                        iDBGR <= iDBGR+1;--increment background counter
                                end if;
                        end if;
                end if;
        end process D_COUNT;
end BEHAVIORAL_DESCRIPTION;
```

Background_Counter_(V=4)

File: background_counter_v4.vhd

```
entity BACKGROUND_COUNTER_V4 is
        generic(NUMDBGR: INTEGER);
--NUMDBGR: number of backgrounds in the first half of the background matrix
        port(CLK        : in BIT;                       --system clock
        RESET           : in BIT;                       --BIST controller reset
        NEXT_DBGR       : in BIT;                       --increment DBGR input
        LAST_DBGR       : out BIT;                      --last data background signal
        DBGR            : out INTEGER range 0 to NUMDBGR-1;
        DBGRC           : out BIT;
        ADBGR           : out INTEGER range 0 to NUMDBGR-1;--background number for N bit
        ADBGRC          : out BIT);                     --ADBGRC for N bit
end BACKGROUND_COUNTER_V4;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_COUNTER_V4 is
        signal iDBGR    : INTEGER range 0 to NUMDBGR-1;     --internal DBGR signal
        signal iDBGRC   : BIT;                              --internal DBGRC signal
        signal iADBGR   : INTEGER range 0 to NUMDBGR-1;     --internal ADBGR signal
        signal iADBGRC: BIT;                                --internal ADBGRC signal
begin
        DBGR<=iDBGR;
        DBGRC<=iDBGRC;
        ADBGR<=iADBGR;
        ADBGRC<=iADBGRC;
        P_INCREMENT: process (iDBGR,iDBGRC)
        begin
                if iDBGR = NUMDBGR-1 then
                        iADBGR <= 0;                    --rolls back to 0 after maximum
                else
                        iADBGR <= iDBGR+1 ;             --increment background number
                end if;
                if iDBGR = NUMDBGR-1 then
                        iADBGRC <= not(iDBGRC);         --toggle ADBGRC
                else
                        iADBGRC <= iDBGRC;
                end if;
        end process P_INCREMENT;
        P_LASTDBGR: process(iDBGR,iDBGRC)
        begin
                if iDBGR = NUMDBGR-1 and iDBGRC = '1' then
                        LAST_DBGR <= '1';               --have reached last background
                else
                        LAST_DBGR <= '0';
                end if;
        end process P_LASTDBGR;
```

```
        D_COUNT: process (CLK,RESET,iADBGR,iADBGRC,NEXT_DBGR)
        begin
                if RESET = '0' then                             -- asynchronous reset
                        iDBGR <= 0;
                        iDBGRC <= '0';
                elsif CLK='0' and CLK'event then                -- synchronize to negative clock edge
                        if NEXT_DBGR = '1' then
                                iDBGR <= iADBGR;        --latch DBGR
                                iDBGRC <= iADBGRC;      --latch DBGRC
                        end if;
                end if;
        end process D_COUNT;
end BEHAVIORAL_DESCRIPTION;
```

Background Counter (V=5t)

File: background_counter_v5t.vhd

```
entity BACKGROUND_COUNTER_V5T is
        generic(NUMDBGR1 : INTEGER;NUMDBGR2:INTEGER);
-- NUMDBGR1 : number of backgrounds in Zones I and II of base matrix
-- NUMDBGR2 : number of backgrounds in base matrix

        port(CLK        : in BIT;
        RESET           : in BIT;
        NEXT_DBGR       : in BIT;               --increment background number
        LAST_DBGR       : out BIT;              --last data background signal
        DBGR1           : out INTEGER range 0 to NUMDBGR1;
        DBGR2           : out INTEGER range 0 to NUMDBGR2;
        DBGRT           : out BIT;              --tranpose matrix signal
        DBGRC           : out BIT;              --complement matrix signal
        ADBGR1          : out INTEGER range 0 to NUMDBGR1;--unlatched DBGR1
        ADBGR2          : out INTEGER range 0 to NUMDBGR2;--unlatched DBGR2
        ADBGRT          : out BIT;              --unlatched DBGRT
        ADBGRC          : out BIT);             --unlatched DBGRC
end BACKGROUND_COUNTER_V5T;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_COUNTER_V5T is
        signal iDBGR1   : INTEGER range 0 to NUMDBGR1;
        signal iDBGR2   : INTEGER range 0 to NUMDBGR2;
        signal iDBGRT   : BIT;
        signal iDBGRC   : BIT;
        signal iADBGR1 : INTEGER range 0 to NUMDBGR1;
        signal iADBGR2 : INTEGER range 0 to NUMDBGR2;
        signal iADBGRT : BIT;
        signal iADBGRC : BIT;
begin
        DBGR1<=iDBGR1;
        DBGR2<=iDBGR2;
        DBGRT<=iDBGRT;
        DBGRC<=iDBGRC;
        ADBGR1<=iADBGR1;
        ADBGR2<=iADBGR2;
        ADBGRT<=iADBGRT;
        ADBGRC<=iADBGRC;
        P_INCREMENT: process (iDBGR1,iDBGR2,iDBGRT,iDBGRC,iADBGR2)
        begin
                if iDBGR1 = NUMDBGR1 then
                        iADBGR1 <= iADBGR2+1;           --ADBGR1 starts from DBGR2+1
                else
                        iADBGR1 <= iDBGR1+1 ;           -- ADBGR1+1
                end if;
                if iDBGR2 = NUMDBGR2 and iDBGR1 = NUMDBGR1 then
                        iADBGR2 <= 0;                   --ADBGR2 starts from 0
                elsif iDBGR2 < NUMDBGR2 and iDBGR1 = NUMDBGR1 then
                        iADBGR2 <= iDBGR2 + 1;          -- ADBGR2+1
                else
                        iADBGR2 <= iDBGR2;
                end if;
```

```
                if iDBGR2 = NUMDBGR2 and iDBGR1 = NUMDBGR1 then
                        iADBGRT <= not(iDBGRT);         --toggle ADBGRT
                else
                        iADBGRT <= iDBGRT;
                end if;
                if iDBGRT = '1' and iDBGR2 = NUMDBGR2 and iDBGR1 = NUMDBGR1 then
                        iADBGRC <= not(iDBGRC);         --toggle ADBGRC
                else
                        iADBGRC <= iDBGRC;
                end if;
        end process P_INCREMENT;
        P_LASTDBGR: process(iDBGR1,iDBGR2,iDBGRT,iDBGRC)
        begin
                if iDBGR1 = NUMDBGR1 and iDBGR2 = NUMDBGR2 and iDBGRT = '1'
                and iDBGRC = '1' then
                        LAST_DBGR <= '1';               -- have reached last background
                else
                        LAST_DBGR <= '0';
                end if;
        end process P_LASTDBGR;
        D_COUNT: process (CLK,RESET,iADBGR1,iADBGR2,iADBGRT,iADBGRC,NEXT_DBGR)
        begin
                if RESET = '0' then                     --asynchrounous reset
                        iDBGR1 <= 1;
                        iDBGR2 <= 0;
                        iDBGRT <= '0';
                        iDBGRC <= '0';
                elsif CLK='0' and CLK'event then
                        if NEXT_DBGR = '1' then
                                iDBGR1 <= iADBGR1;      --latch all outputs
                                iDBGR2 <= iADBGR2;
                                iDBGRT <= iADBGRT;
                                iDBGRC <= iADBGRC;
                        end if;
                end if;
        end process D_COUNT;
end BEHAVIORAL_DESCRIPTION;
```

Background_Counter (V=5x)

File: background_counter_v5x.vhd

```
entity BACKGROUND_COUNTER_V5X is
        generic(NUMDBGR1 : INTEGER );          --number of backgrounds in base matrix
        port(CLK         : in BIT;
        RESET            : in BIT;
        NEXT_DBGR        : in BIT;             --increment background number
        LAST_DBGR        : out BIT;            -- last data background signal
        DBGR1            : out INTEGER range 0 to NUMDBGR1;
        DBGR2            : out INTEGER range 0 to NUMDBGR1;
        DBGRC            : out BIT;
        ADBGR1           : out INTEGER range 0 to NUMDBGR1;
        ADBGR2           : out INTEGER range 0 to NUMDBGR1;
        ADBGRC           : out BIT);
end BACKGROUND_COUNTER_V5X;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_COUNTER_V5X is
        signal iDBGR1    : INTEGER range 0 to NUMDBGR1;
        signal iDBGR2    : INTEGER range 0 to NUMDBGR1;
        signal iDBGRC    : BIT;
        signal iADBGR1 : INTEGER range 0 to NUMDBGR1;
        signal iADBGR2 : INTEGER range 0 to NUMDBGR1;
        signal iADBGRC : BIT;
begin
        DBGR1<=iDBGR1;
        DBGR2<=iDBGR2;
        DBGRC<=iDBGRC;
        ADBGR1<=iADBGR1;
        ADBGR2<=iADBGR2;
        ADBGRC<=iADBGRC;

        P_INCREMENT: process (iDBGR1,iDBGR2,iDBGRC)
        begin
                if iDBGR1 = NUMDBGR1 then
                        iADBGR1 <= 0;                   --ADBGR1 starts from 0
                else
                        iADBGR1 <= iDBGR1+1 ;           --ADBGR1+1
                end if;
                if iDBGR2 = NUMDBGR1 and iDBGR1 = NUMDBGR1 then
                        iADBGR2 <= 0;                   --ADBGR2 starts from 0
                elsif iDBGR2 < NUMDBGR1 and iDBGR1 = NUMDBGR1 then
                        iADBGR2 <= iDBGR2 + 1;          -- ADBGR2+1
                else
                        iADBGR2 <= iDBGR2;
                end if;
                if iDBGR2 = NUMDBGR1 and iDBGR1 = NUMDBGR1 then
                        iADBGRC <= not(iDBGRC);         --toggle ADBGRC
                else
                        iADBGRC <= iDBGRC;
                end if;
        end process P_INCREMENT;
```

```
        P_LASTDBGR: process(iDBGR1,iDBGR2,iDBGRC)
        begin
                if iDBGR1 = NUMDBGR1 and iDBGR2 = NUMDBGR1 and iDBGRC = '1' then
                        LAST_DBGR <= '1';               --have reached last background
                else
                        LAST_DBGR <= '0';
                end if;
        end process P_LASTDBGR;

        D_COUNT: process (CLK,RESET,iADBGR1,iADBGR2,iADBGRC,NEXT_DBGR)
        begin
                if RESET = '0' then
                        iDBGR1 <= 0;                    --reset outputs to 0
                        iDBGR2 <= 0;
                        iDBGRC <= '0';
                elsif CLK='0' and CLK'event then
                        if NEXT_DBGR = '1' then
                                iDBGR1 <= iADBGR1;      --latch ouputs
                                iDBGR2 <= iADBGR2;
                                iDBGRC <= iADBGRC;
                        end if;
                end if;
        end process D_COUNT;
end BEHAVIORAL_DESCRIPTION;
```

Background Code Logic (V=2)

File: bg2.vhd

```
-- Background matrix for V-2 has two columns, all-zero column and all-one column, therefore P is
-- never equal to N

entity BACKGROUND_LOGIC_V2 is
        port(APEQN : out BIT;                   --P=N ouput
        PEQN : out BIT);                        --unlatched P=N output
end BACKGROUND_LOGIC_V2;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_LOGIC_V2 is
begin
        APEQN <= '0';                           --AP=N always 0
        PEQN <='0';                             --P=N always 0
end BEHAVIORAL_DESCRIPTION;
```

Background Code Logic (V=3)

File: bg3.vhd

```
use work.CONVERT_TYPE.all;

entity BACKGROUND_LOGIC_V3 is
        generic(N:INTEGER;M:INTEGER;NUMDBGR:INTEGER;S:INTEGER);
-- N: size of RAM
-- M: log2(N), width of address bus
-- NUMDBGR: number of backgrounds
-- S: log2(M) number of bits to represent the number of zeros in the address bits.

        port( CLK       : in BIT;                          -- system clock
        RESET           : in BIT;                          -- controller reset
        ADDRESS         : in INTEGER range 0 to N-1;       -- address bus
        DBGR            : in INTEGER range 0 to NUMDBGR-1; -- background number
        NEXT_ADDR       : in BIT;                          -- next address signal
        APEQN           : out BIT;                         -- unlatched P=N signal
        PEQN            : out BIT);                        -- latched P=N signal
end BACKGROUND_LOGIC_V3;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_LOGIC_V3 is

        signal C        : BIT_VECTOR(0 to NUMDBGR);
        signal ADDR     : BIT_VECTOR(M-1 downto 0);
        signal SUMZERO: INTEGER range 0 to M;
        signal SUM0     : BIT_VECTOR(S-1 downto 0);
        signal AN       : BIT;
        signal AP       : BIT;
        signal iApeqN   : Bit ;
        component SUMZEROLOGIC                          --use SUMZEROLOGIC component
                generic(M:INTEGER);
                port(ADDR       : in BIT_VECTOR(M-1 downto 0);
                NUMZEROS        : out INTEGER range 0 to M);
        end component;
begin
        ADDR <= INTEGER_TO_BIT_VECTOR(ADDRESS,M);       --convert address from integer
                                                        --to bit vector
        U1: SUMZEROLOGIC                                --instantiate SUMZEROLOGIC to count
                                                        --the number of zeros
        generic map(M =>M)                              --address bus ADDR
        port map (ADDR => ADDR,
                NUMZEROS => SUMZERO);                   --return number of zeros in integer
        SUM0 <= INTEGER_TO_BIT_VECTOR(SUMZERO,S);--convert number of zeros to bit vector
        C <= '0'& ADDR & '1' & SUM0 & '0';              --concatenate the bits in the order
                                                        -- '0',ADDR,'1',SUMZERO,'0' into C
        AP<=C(DBGR);                                    --unlatched P bit
        AN<=C(DBGR+1);                                  --unlatched N bit
        iAPEQN <= not(AP xor AN);                       --iApeqN = 1 if AP=AN
        APEQN <= iAPEQN;
```

```
        P_DBGR: process (CLK,RESET,NEXT_ADDR,iAPEQN)
        begin
                if RESET = '0' then                             --asynchronous set
                        PEQN <= '1';
                elsif CLK = '0' and CLK'event then              --negative clock edge
                        if NEXT_ADDR = '1' then
                                PEQN <= iAPEQN;                 --latch P=N
                        end if;
                end if;
        end process P_DBGR;
end Behavioral_description;
```

SUMZERO Logic

File: sum0logic.vhd

```
-- use sumzero3 and sumzero2 to find the number of zeros in M-bit address bus
-- output from the sumzero cells are input into a tree structure adder

entity SUMZEROLOGIC is
        generic(M:INTEGER);
        port(ADDR : in BIT_VECTOR(M-1 downto 0);        --M-bit address
        NUMZEROS    : out INTEGER range 0 to M);        --number of zeros
end SUMZEROLOGIC;

architecture  BEHAVIORAL_DESCRIPTION of SUMZEROLOGIC is
        signal TEMPZERO1        : INTEGER range 0 to 3;
        signal TEMPZERO2        : INTEGER range 0 to 3;
        signal TEMPZERO3        : INTEGER range 0 to 3;
        signal TEMPZERO4        : INTEGER range 0 to 3;
        signal TEMPZERO5        : INTEGER range 0 to 3;
        signal TEMPZERO6        : INTEGER range 0 to 3;
        signal TEMPZERO7        : INTEGER range 0 to 3;
        signal TEMPZERO8        : INTEGER range 0 to 3;
        signal TEMPZERO9        : INTEGER range 0 to 3;
        signal TEMPZERO1a       : INTEGER range 0 to 2;
        signal TEMPZERO2a       : INTEGER range 0 to 2;
        signal TEMPZERO3a       : INTEGER range 0 to 2;
        signal TEMPZERO4a       : INTEGER range 0 to 2;
        signal TEMPZERO5a       : INTEGER range 0 to 2;
        signal TEMPZERO6a       : INTEGER range 0 to 2;
        signal TEMPZERO7a       : INTEGER range 0 to 2;
        signal TEMPZERO8a       : INTEGER range 0 to 2;
        signal TEMPZERO9a       : INTEGER range 0 to 2;
component SUMZERO3
        port ( I : in BIT_VECTOR(2 downto 0);
        Q : out INTEGER range 0 to 3);
end component;
component SUMZERO2
        port ( I : in BIT_VECTOR(1 downto 0);
        Q : out INTEGER range 0 to 2);
end component;
begin
M2:
if M=2 generate
        U1: SUMZERO2
        port map ( I => ADDR,
                   Q => NUMZEROS);
end generate;
M3:
if M=3 generate
U1: SUMZERO3
        port map ( I => ADDR,
                   Q => NUMZEROS);
end generate;
M4:
if M=4 generate
U1: SUMZERO2
        port map ( I => ADDR(1 downto 0),
                   Q => TEMPZERO1a);
```

```
U2: SUMZERO2
        port map ( I => ADDR(3 downto 2),
                   Q => TEMPZERO2a);
        NUMZEROS <= TEMPZERO1a + TEMPZERO2a;
end generate;
M5:
if M=5 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO2
        port map ( I => ADDR(4 downto 3),
                   Q => TEMPZERO2a);
        NUMZEROS <= TEMPZERO1 + TEMPZERO2a;
end generate;
M6:
if M=6 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
        Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
        NUMZEROS <= TEMPZERO1 + TEMPZERO2;
end generate;
M7:
if M=7 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO2
        port map ( I => ADDR(4 downto 3),
                   Q => TEMPZERO2a);
U3: SUMZERO2
        port map ( I => ADDR(6 downto 5),
        Q => TEMPZERO3a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2a) + TEMPZERO3a;
end generate;
M8:
if M=8 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO2
        port map ( I => ADDR(7 downto 6),
                   Q => TEMPZERO3a);
        NUMZEROS <= TEMPZERO1 + TEMPZERO2 + TEMPZERO3a;
end generate;
M9:
if M=9 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
```

```
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + TEMPZERO3;
end generate;
M10:
if M=10 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO2
        port map ( I => ADDR(7 downto 6),
                   Q => TEMPZERO3a);
U4: SUMZERO2
        port map ( I => ADDR(9 downto 8),
                   Q => TEMPZERO4a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + (TEMPZERO3a + TEMPZERO4a);
end generate;
M11:
if M=11 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO2
        port map ( I => ADDR(10 downto 9),
                   Q => TEMPZERO4a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4a);
end generate;
M12:
if M=12 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4);
end generate;
```

```
M13:
if M=13 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO2
        port map ( I => ADDR(10 downto 9),
                   Q => TEMPZERO4a);
U5: SUMZERO2
        port map ( I => ADDR(12 downto 11),
                   Q => TEMPZERO5a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4a) +
TEMPZERO5a);
end generate;
M14:
if M=14 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO2
        port map ( I => ADDR(13 downto 12),
                   Q => TEMPZERO5a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4) +
TEMPZERO5a);
end generate;
M15:
if M=15 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
```

```
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4) +
TEMPZERO5);
end generate;
M16:
if M=16 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO2
        port map ( I => ADDR(13 downto 12),
                   Q => TEMPZERO5a);
U6: SUMZERO2
        port map ( I => ADDR(15 downto 14),
                   Q => TEMPZERO6a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4) +
(TEMPZERO5a + TEMPZERO6a));
end generate;
M17:
if M=17 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO2
        port map ( I => ADDR(16 downto 15),
                   Q => TEMPZERO6a);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4) +
(TEMPZERO5 + TEMPZERO6a));
end generate;
M18:
if M=18 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
```

```
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
        NUMZEROS <= (TEMPZERO1 + TEMPZERO2) + ((TEMPZERO3 + TEMPZERO4) +
(TEMPZERO5 + TEMPZERO6));
end generate;
M19:
if M=19 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO2
        port map ( I => ADDR(16 downto 15),
                   Q => TEMPZERO6a);
U7: SUMZERO2
        port map ( I => ADDR(18 downto 17),
                   Q => TEMPZERO7a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6a) + TEMPZERO7a);
end generate;
M20:
if M=20 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
```

```
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
U7: SUMZERO2
        port map ( I => ADDR(19 downto 18),
                   Q => TEMPZERO7a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6) + TEMPZERO7a);
end generate;
M21:
if M=21 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
U7: SUMZERO3
        port map ( I => ADDR(20 downto 18),
                   Q => TEMPZERO7);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6) + TEMPZERO7);
end generate;
M22:
if M=22 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
```

```
U7: SUMZERO2
        port map ( I => ADDR(19 downto 18),
                   Q => TEMPZERO7a);
U8: SUMZERO2
        port map ( I => ADDR(21 downto 20),
                   Q => TEMPZERO8a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6) + (TEMPZERO7a + TEMPZERO8a));
end generate;
M23:
if M=23 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
U7: SUMZERO3
        port map ( I => ADDR(20 downto 18),
                   Q => TEMPZERO7);
U8: SUMZERO2
        port map ( I => ADDR(22 downto 21),
                   Q => TEMPZERO8a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6) + (TEMPZERO7 + TEMPZERO8a));
end generate;
M24:
if M=24 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
```

```
U7: SUMZERO3
        port map ( I => ADDR(20 downto 18),
                   Q => TEMPZERO7);
U8: SUMZERO3
        port map ( I => ADDR(23 downto 21),
                   Q => TEMPZERO8);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
((TEMPZERO5 + TEMPZERO6) + (TEMPZERO7 + TEMPZERO8));
end generate;
M25:
if M=25 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
U7: SUMZERO3
        port map ( I => ADDR(20 downto 18),
                   Q => TEMPZERO7);
U8: SUMZERO2
        port map ( I => ADDR(22 downto 21),
                   Q => TEMPZERO8a);
U9: SUMZERO2
        port map ( I => ADDR(24 downto 23),
                   Q => TEMPZERO9a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
(((TEMPZERO5 + TEMPZERO6) + (TEMPZERO7 + TEMPZERO8a)) + TEMPZERO9a);
end generate;
M26:
if M=26 generate
U1: SUMZERO3
        port map ( I => ADDR(2 downto 0),
                   Q => TEMPZERO1);
U2: SUMZERO3
        port map ( I => ADDR(5 downto 3),
                   Q => TEMPZERO2);
U3: SUMZERO3
        port map ( I => ADDR(8 downto 6),
                   Q => TEMPZERO3);
U4: SUMZERO3
        port map ( I => ADDR(11 downto 9),
                   Q => TEMPZERO4);
U5: SUMZERO3
        port map ( I => ADDR(14 downto 12),
                   Q => TEMPZERO5);
```

```
U6: SUMZERO3
        port map ( I => ADDR(17 downto 15),
                   Q => TEMPZERO6);
U7: SUMZERO3
        port map ( I => ADDR(20 downto 18),
                   Q => TEMPZERO7);
U8: SUMZERO3
        port map ( I => ADDR(23 downto 21),
                   Q => TEMPZERO8);
U9: SUMZERO2
        port map ( I => ADDR(25 downto 22),
                   Q => TEMPZERO9a);
        NUMZEROS <= ((TEMPZERO1 + TEMPZERO2) + (TEMPZERO3 + TEMPZERO4)) +
(((TEMPZERO5 + TEMPZERO6) + (TEMPZERO7 + TEMPZERO8)) + TEMPZERO9a);
end generate;
end BEHAVIORAL_DESCRIPTION;
```

SUMZERO3 and SUMZERO2

File: sumzero.vhd

```
-- 3-bit sumzero cell
-- output the number of zeros in the 3-bit input

entity SUMZERO3 is
        port (I : in BIT_VECTOR (2 downto 0);      --3-bit input
        Q : out INTEGER range 0 to 3);             --number of zeros
        end SUMZERO3;
architecture BEHAVIORAL_DESCRIPTION of SUMZERO3 is
begin
P_SUMZ: process(I)
        begin
                if I = "000" then Q <= 3;
                elsif I = "001" or I = "010" or I = "100" then Q <= 2;
                elsif I = "011" or I = "101" or I = "110" then Q <= 1;
                elsif I = "111" then Q <= 0;
                end if;
        end process P_SUMZ;
end BEHAVIORAL_DESCRIPTION;



-- 2-bit sumzero cell
-- output the number of zeros in the 2-bit input

entity SUMZERO2 is
        port (I : in BIT_VECTOR (1 downto 0);      --2-bit input
        Q : out INTEGER range 0 to 2);             --number of zeros
end SUMZERO2;
architecture BEHAVIORAL_DESCRIPTION of SUMZERO2 is
begin
P_SUMZ: process(I)
        begin
                if I = "00" then Q <= 2;
                elsif I = "01" or I = "10" then Q <= 1;
                elsif I = "11" then Q <= 0;
                end if;
        end process P_SUMZ;
end BEHAVIORAL_DESCRIPTION;
```

Background Code Logic (V=4)

File: bg4.vhd

```
use work.CONVERT_TYPE.all;
use work.XORCOL.all;

entity BACKGROUND_LOGIC_V4 is
        generic (NUMXOR:INTEGER;N:INTEGER;M:INTEGER;NUMDBGR:INTEGER);
-- NUMXOR: number of address pairs xor-ed together
-- N: size of RAM
-- M: log2(N), width of address bus
-- NUMDBGR: number of backgrounds in the first half of background matrix

        port(CLK        : in BIT;
        RESET           : in BIT;
        ADDRESS         : in INTEGER range 0 to N-1;
        DBGR            : in INTEGER range 0 to NUMDBGR-1;
        DBGRC           : in BIT;
        ADBGR           : in INTEGER range 0 to NUMDBGR-1;    --unlatched DBGR
        ADBGRC          : in BIT;                             --unlatched DBGRC
        NEXT_ADDR       : in BIT;
        PEQN            : out BIT;
        APEQN           : out BIT);
end BACKGROUND_LOGIC_V4;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_LOGIC_V4 is
        signal C        : BIT_VECTOR(0 TO NUMDBGR);
        signal ADDR     : BIT_VECTOR(M-1 downto 0);
        signal AN       : BIT;
        signal AP       : BIT;
        signal iApeqN   : BIT;
begin

        ADDR <= INTEGER_TO_BIT_VECTOR(ADDRESS,M); --convert address to bit vector
        C(0) <= '0';                                    --1st field: all-zero column
        C(1 to M) <= ADDR;                              --2nd field: address
        C(M+1 to NUMXOR + M) <= GEN_XORCOL(M,NUMXOR,ADDR);--3rd field: xor
                                                        --all pairs of address bits
        AP <= DBGRC xor C(DBGR);                        --unlatched P bit
        AN <= ADBGRC xor C(ADBGR);                      --unlatched N bit

        iAPEQN <= not(AP xor AN);
        APEQN <= iAPEQN;
        P_DBGR: process (CLK,RESET,NEXT_ADDR,iAPEQN)
        begin
                if RESET = '0' then                     --asynchronous set
                        PEQN <= '1';
                elsif CLK = '0' and CLK'event then
                        if NEXT_ADDR = '1' then
                                PEQN <= iAPEQN;         --latched P=N
                        end if;
                end if;
        end process P_DBGR;

end BEHAVIORAL_DESCRIPTION;
```

Function GEN_XOR_COL

File: xorcol.vhd

```
-- Library XORCOL contains the GEN_XORCOL function
-- This functions generates the third field in an (n,3)-exhaustive code.
-- All the pairs of address lines are xor-ed together.

package XORCOL is
        function GEN_XORCOL(M,NUMXOR:INTEGER;ADDR:BIT_VECTOR)
-- M: width of address bus
-- NUMXOR: number of pairs of address lines
-- ADDR : address bus
        return BIT_VECTOR;
end XORCOL;

package body XORCOL is
        function GEN_XORCOL(M,NUMXOR:INTEGER;ADDR:BIT_VECTOR)
        return BIT_VECTOR is
                variable INDEX : INTEGER;
                variable TEMP_VECTOR : BIT_VECTOR(0 to NUMXOR-1);
                begin
                INDEX := NUMXOR-1;
                for I in 0 to M-2 loop                  --generates all pairs of address lines
                        for J in I+1 to M-1 loop
                                TEMP_VECTOR(INDEX) := ADDR(I) xor ADDR(J);--xor the two lines
                                INDEX := INDEX - 1;
                        end loop;
                end loop;
                return TEMP_VECTOR;
        end;
end XORCOL;
```

Background Code Logic (V=5t)

File: bg5t.vhd

```
use work.CONVERT_TYPE.all;
use work.XORCOL.all;

entity BACKGROUND_LOGIC_V5T is
        generic(N:INTEGER;M:INTEGER;NUMXOR:INTEGER;NUMDBGR1:INTEGER;
        NUMDBGR2:INTEGER);
-- N: size of RAM
-- M: log2(N), width of address bus
-- NUMXOR: number of pairs from 1/2 of the address lines
-- NUMDBGR1: number of backgrounds in Zones I and II of base matrix
-- NUMDBGR2: number of backgrounds in base matrix
        port(CLK        : in BIT;
        RESET           : in BIT;
        ADDRESS         : in INTEGER range 0 to N-1;
        DBGR1           : in INTEGER range 0 to NUMDBGR1;
        DBGR2           : in INTEGER range 0 to NUMDBGR2;
        DBGRT           : in BIT;
        DBGRC           : in BIT;
        ADBGR1          : in INTEGER range 0 to NUMDBGR1;
        ADBGR2          : in INTEGER range 0 to NUMDBGR2;
        ADBGRT          : in BIT;
        ADBGRC          : in BIT;
        NEXT_ADDR       : in BIT;
        APEQN           : out BIT;
        PEQN            : out BIT);
end BACKGROUND_LOGIC_V5T;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_LOGIC_V5T is
        signal P1       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal P2       :  BIT_VECTOR(0 TO NUMDBGR2 );
        signal N1       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal N2       :  BIT_VECTOR(0 TO NUMDBGR2 );
        signal ADDR     : BIT_VECTOR(M-1 downto 0);
        signal ADDR1    : BIT_VECTOR(M/2-1 downto 0);
        signal ADDR2    : BIT_VECTOR(M/2-1 downto 0);
        signal N_ADDR1: BIT_VECTOR(M/2-1 downto 0);
        signal N_ADDR2: BIT_VECTOR(M/2-1 downto 0);
        signal AP       : BIT;
        signal AN       : BIT;
        signal iAPEQN   : BIT;
begin
        ADDR <= INTEGER_TO_BIT_VECTOR(ADDRESS,M); --convert address to bit vector
        P_TRANPOSE: process (DBGRT,ADDR)                --transpose xor matrix for P bit
        begin
                if DBGRT = '0' then
                        ADDR1 <= ADDR(M/2 -1 downto 0);         --ADDR1=column address
                        ADDR2 <= ADDR(M-1 downto M/2);          --ADDR2=row address
                else
                        ADDR1 <= ADDR(M-1 downto M/2);          --ADDR1=row address
                        ADDR2 <= ADDR(M/2 -1 downto 0);         --ADDR2=column address
                end if;
        end process P_TRANPOSE;
        P1(0) <= '0';                           --Zone I (1/2 (sqrt(n),3)-exhaustive code)
        P1(1 to M/2) <= ADDR1;                  --Zone II
```

```
        P1(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,ADDR1); --Zone III
        P2(0) <='0';                                    --Zone I (1/2 (sqrt(n),2)-exhaustive code)
        P2(1 to M/2) <= ADDR2;                          --Zone II
        AP <=  DBGRC xor P1(DBGR1) xor P2(DBGR2);--unlatched P bit
        N_TRANPOSE: process (ADBGRT,ADDR)               --transpose xor matrix for N bit
        begin
                if ADBGRT = '0' then
                        N_ADDR1 <= ADDR(M/2 -1 downto 0);    --N_ADDR1=column address
                        N_ADDR2 <= ADDR(M-1 downto M/2);     --N_ADDR2=row address
                else
                        N_ADDR1 <= ADDR(M-1 downto M/2);     --N_ADDR1=row address
                        N_ADDR2 <= ADDR(M/2 -1 downto 0);    --N_ADDR2=column address
                end if;
        end process N_TRANPOSE;
        N1(0) <= '0';                                   --Zone I (1/2 (sqrt(n),3)-exhaustive code)
        N1(1 to M/2) <= N_ADDR1;                        --Zone II
        N1(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,N_ADDR1);--Zone III
        N2(0) <='0';                                                --Zone I (1/2 (sqrt(n),2)-exhaustive code)
        N2(1 to M/2) <= N_ADDR2;        --Zone II
        AN <=  ADBGRC xor N1(ADBGR1) xor N2(ADBGR2);-- unlatched N bit
        iAPEQN <= not(AP xor AN);
        APEQN <= iAPEQN;
        P_DBGR: process(CLK,RESET,NEXT_ADDR,iAPEQN)
        begin
                if RESET = '0' then
                        PEQN <= '1';
                elsif CLK = '0' and CLK'event then
                        if NEXT_ADDR = '1' then
                                PEQN <= iAPEQN;
                        end if;
                end if;
        end process P_DBGR;
end BEHAVIORAL_DESCRIPTION;
```

Background Code Logic (V=5x)

File: bg5x.vhd

```
use work.CONVERT_TYPE.all;
use work.XORCOL.all;

entity BACKGROUND_LOGIC_V5X is
        generic(N:INTEGER;M:INTEGER;NUMXOR:INTEGER;NUMDBGR1:INTEGER);
--N: size of RAM
--M: log2(N), width of address bus
--NUMXOR: number of pairs from 1/2 of the address lines
--NUMDBGR1: number of backgrounds in base matirx

        port(CLK        : in BIT;
        RESET           : in BIT;
        ADDRESS         : in INTEGER range 0 to N-1;
        DBGR1           : in INTEGER range 0 to NUMDBGR1;
        DBGR2           : in INTEGER range 0 to NUMDBGR1;
        DBGRC           : in BIT;
        ADBGR1          : in INTEGER range 0 to NUMDBGR1;
        ADBGR2          : in INTEGER range 0 to NUMDBGR1;
        ADBGRC          : in BIT;
        NEXT_ADDR       : in BIT;
        APEQN           : out BIT;
        PEQN            : out BIT);
end BACKGROUND_LOGIC_V5X;

architecture BEHAVIORAL_DESCRIPTION of BACKGROUND_LOGIC_V5X is
        signal P1       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal P2       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal N1       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal N2       :  BIT_VECTOR(0 TO NUMDBGR1);
        signal ADDR     : BIT_VECTOR(M-1 downto 0);
        signal ADDR1    : BIT_VECTOR(M/2-1 downto 0);
        signal ADDR2    : BIT_VECTOR(M/2-1 downto 0);
        signal N_ADDR1 : BIT_VECTOR(M/2-1 downto 0);
        signal N_ADDR2 : BIT_VECTOR(M/2-1 downto 0);
        signal AP       : BIT;
        signal AN       : BIT;
        signal iAPEQN   : BIT;
begin
        ADDR <= INTEGER_TO_BIT_VECTOR(ADDRESS,M);       --convert address to bit vector
        ADDR1 <= ADDR(M/2 -1 downto 0);                 --ADDR1=column address
        ADDR2 <= ADDR(M-1 downto M/2);                  --ADDR2=row address
        P1(0) <= '0';                                   --Zone I (1/2 (sqrt(n),3)-exhaustive code)
        P1(1 to M/2) <= ADDR1;                          --Zone II
        P1(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,ADDR1);--Zone III
        P2(0) <='0';                                    --Zone I (1/2 (sqrt(n),3)-exhaustive code)
        P2(1 to M/2) <= ADDR2;                          --Zone II
        P2(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,ADDR2);--Zone III
        AP <= DBGRC xor P1(DBGR1) xor P2(DBGR2);        -- unlatched P

        N_ADDR1 <= ADDR(M/2 -1 downto 0);               --N_ADDR1=column address (for N bit)
        N_ADDR2 <= ADDR(M-1 downto M/2);                --N_ADDR2=row address (for N bit)
        N1(0) <= '0';                                   --Zone I (1/2 (sqrt(n),3)-exhaustive code)
        N1(1 to M/2) <= N_ADDR1;                        --Zone II
        N1(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,N_ADDR1);--Zone III
```

```
        N2(0) <='0';
        N2(1 to M/2) <= N_ADDR2;
        N2(M/2+1 to NUMXOR+M/2) <= GEN_XORCOL(M/2,NUMXOR,N_ADDR2);
        AN <=  ADBGRC xor N1(ADBGR1) xor N2(ADBGR2); --unlatched N
        iAPEQN <= not(AP xor AN);
        APEQN <= iAPEQN;
        P_DBGR: process(CLK,RESET,NEXT_ADDR,iAPEQN)
        begin
                if RESET = '0' then
                        PEQN <= '1';
                elsif CLK = '0' and CLK'event then
                        if NEXT_ADDR = '1' then
                                PEQN <= iAPEQN;
                        end if;
                end if;
        end process P_DBGR;
end BEHAVIORAL_DESCRIPTION;
```

Response_Analyzer

File: resp_anlyz.vhd

```
use work.BTYPES.all;

entity RESPONSE_ANALYZER is
        generic(LFSRDEG:INTEGER);
        port(CLK        : in BIT;          -- system clock
        RESET           : in BIT;          -- BIST controller reset
        CLRSIG          : in BIT;          -- clear LFSR input
        ENABLE          : in BIT;          -- enable LFSR input
        LOADREG1        : in BIT;          -- load register 1 input
        LOADREG2        : in BIT;          -- load register 2 input
        B               : in BIT;          -- RAM data input
        PEQN            : in BIT;          -- P=N signal
        PHASE2          : in BIT;          -- Phase2 flag
        STATE           : in STATES;       -- state variable
        ASYNCSTATE      : in STATES;       -- next state variable
        PASS            : out BIT);        -- PASS ouput
end RESPONSE_ANALYZER;

architecture STRUCTURE of RESPONSE_ANALYZER is
        signal ALFSR_OUT        : BIT_VECTOR(LFSRDEG-1 downto 0);--unlatched LFSR output
        signal LFSR_OUT         : BIT_VECTOR(LFSRDEG-1 downto 0);--latched LFSR output
        signal REG_OUT1         : BIT_VECTOR(LFSRDEG-1 downto 0);--register 1 content
        signal REG_OUT2         : BIT_VECTOR(LFSRDEG-1 downto 0);--register 2 content
        signal COMP_IN          : BIT_VECTOR(LFSRDEG-1 downto 0);--input to comparator
        signal LFSR_IN          : BIT;                  --input to LFSR (1 bit)
        signal RB               : BIT;                  --output from read bit flipper
        signal PEQNBAR          : BIT;                  -- P=N bar

component READ_BIT_FLIPPER
        port(B          : in BIT;
        STATE           : in STATES;
        PEQNBAR         : in BIT;
        Q               : out BIT);
end component;
component LFSR16
        port(CLK        : in BIT;
        RESET           : in BIT;
        CLRSIG          : in BIT;
        EN              : in BIT;
        LFSR_IN         : in BIT;
        ALFSR           : out BIT_VECTOR(15 downto 0);
        LFSR            : out BIT_VECTOR(15 downto 0));
end component;
component LFSR20
        port(CLK        : in BIT;
        RESET           : in BIT;
        CLRSIG          : in BIT;
        EN              : in BIT;
        LFSR_IN         : in BIT;
        ALFSR           : out BIT_VECTOR(19 downto 0);
        LFSR            : out BIT_VECTOR(19 downto 0));
end component;
```

```
component LFSR25
        port(CLK        : in BIT;
        RESET           : in BIT;
        CLRSIG          : in BIT;
        EN              : in BIT;
        LFSR_IN         : in BIT;
        ALFSR           : out BIT_VECTOR(24 downto 0);
        LFSR            : out BIT_VECTOR(24 downto 0));
end component;
component LFSR30
        port(CLK        : in BIT;
        RESET           : in BIT;
        CLRSIG          : in BIT;
        EN              : in BIT;
        LFSR_IN         : in BIT;
        ALFSR           : out BIT_VECTOR(29 downto 0);
        LFSR            : out BIT_VECTOR(29 downto 0));
end component;
component REGISTERS
        generic(K:INTEGER);
        port( CLK       : in BIT;
        RESET           : in BIT;
        REG_IN          : in BIT_VECTOR(LFSRDEG-1 downto 0);
        LOADREG         : in BIT;
        REG_OUT         : out BIT_VECTOR(LFSRDEG-1 downto 0));
end component;
component COMPARATOR
        generic(K:INTEGER);
        port(A  : in BIT_VECTOR(LFSRDEG-1 downto 0);
        B       : in BIT_VECTOR(LFSRDEG-1 downto 0);
        AEQB    : out BIT);
end component;

begin
PEQNBAR <= not(PEQN);
P_SELREG: process (ASYNCSTATE,REG_OUT1,REG_OUT2)
-- select the source of the input to the comparator
-- if next state = S8, select from register 2
-- else select from register 1
begin
        if ASYNCSTATE = S8 then
                COMP_IN <= REG_OUT2;
        else
                COMP_IN <= REG_OUT1;
        end if;
end process P_SELREG;
P_SELBIT: process (PHASE2,B,RB)
-- select input to LFSR
-- if current phase is not phase 2, select output from read bit flipper
-- else select RAM data
begin
        if PHASE2 = '0' then
                LFSR_IN <= RB;
        else
                LFSR_IN <= B;
        end if;
end process P_SELBIT;
```

```
READBITFLIPPER: READ_BIT_FLIPPER
        port map(B              => B,
                 STATE          => STATE,
                 PEQNBAR        => PEQNBAR,
                 Q              => RB);
SELECT_LFSR16:
if LFSRDEG = 16 generate
        LFSR_16: LFSR16
        port map(CIK            => CLK,
                 RESET          => RESET,
                 CLRSIG         => CLRSIG,
                 EN             => ENABLE,
                 LFSR_IN        => LFSR_IN,
                 ALFSR          => ALFSR_OUT,
                 LFSR           => LFSR_OUT);
end generate;
-- selectively generate LFSR given the LFSR degree LFSRDEG
SELECT_LFSR20:
if LFSRDEG = 20 generate
        LFSR_20: LFSR20
        port map(CIK            => CLK,
                 RESET          => RESET,
                 CLRSIG         => CLRSIG,
                 EN             => ENABLE,
                 LFSR_IN        => LFSR_IN,
                 ALFSR          => ALFSR_OUT,
                 LFSR           => LFSR_OUT);
end generate;
SELECT_LFSR25:
if LFSRDEG = 25 generate
        LFSR_25: LFSR25
        port map(CIK            => CLK,
                 RESET          => RESET,
                 CLRSIG         => CLRSIG,
                 EN             => ENABLE,
                 LFSR_IN        => LFSR_IN,
                 ALFSR          => ALFSR_OUT,
                 LFSR           => LFSR_OUT);
end generate;
SELECT_LFSR30:
if LFSRDEG = 30 generate
        LFSR_30: LFSR30
        port map(CIK            => CLK,
                 RESET          => RESET,
                 CLRSIG         => CLRSIG,
                 EN             => ENABLE,
                 LFSR_IN        => LFSR_IN,
                 ALFSR          => ALFSR_OUT,
                 LFSR           => LFSR_OUT);
end generate;
REGISTER1: REGISTERS
        generic map(K=>LFSRDEG)
        port map(CLK            => CLK,
                 RESET          => RESET,
                 REG_IN         => ALFSR_OUT,
                 LOADREG        => LOADREG1,
                 REG_OUT        => REG_OUT1);
```

```
REGISTER2: REGISTERS
        generic map(K=>LFSRDEG)
        port map(CLK            => CLK,
                 RESET          => RESET,
                 REG_IN         => ALFSR_OUT,
                 LOADREG        => LOADREG2,
                 REG_OUT        => REG_OUT2);

COMPARATORS: COMPARATOR
        generic map(K=>LFSRDEG)
        port map(A      =>ALFSR_OUT,
                 B      =>COMP_IN,
                 AeqB   =>PASS);
end STRUCTURE;
```

LFSR (16 bit)

File: lfsr16.vhd

```
entity LFSR16 is
        port(CLK        : in BIT;           -- system clock
        RESET           : in BIT;           -- BIST controller reset
        CLRSIG          : in BIT;           -- clear LFSR signal
        EN              : in BIT;           -- LFSR enable signal
        LFSR_IN         : in BIT;           -- LFSR input, 1 bit wide
        ALFSR           : out BIT_VECTOR(15 downto 0);          -- unlatched LFSR output
        LFSR            : out BIT_VECTOR(15 downto 0));         -- latched LFSR output
end LFSR16;

architecture BEHAVIORAL_DESCRIPTION of LFSR16 is
        signal LFSR_OUT: BIT_VECTOR(15 downto 0);
        signal ALFSR_OUT : BIT_VECTOR(15 downto 0);
begin
        ALFSR <= ALFSR_OUT;
        LFSR <= LFSR_OUT;
        ALFSR_OUT(0) <= LFSR_IN xor LFSR_OUT(15);
        ALFSR_OUT(1) <= LFSR_OUT(0);
        ALFSR_OUT(2) <= LFSR_OUT(1) xor LFSR_OUT(15);
        ALFSR_OUT(3) <= LFSR_OUT(2) xor LFSR_OUT(15);
        ALFSR_OUT(4) <= LFSR_OUT(3);
        ALFSR_OUT(5) <= LFSR_OUT(4) xor LFSR_OUT(15);
        ALFSR_OUT(6) <= LFSR_OUT(5);
        ALFSR_OUT(7) <= LFSR_OUT(6);
        ALFSR_OUT(8) <= LFSR_OUT(7);
        ALFSR_OUT(9) <= LFSR_OUT(8);
        ALFSR_OUT(10) <= LFSR_OUT(9);
        ALFSR_OUT(11) <= LFSR_OUT(10);
        ALFSR_OUT(12) <= LFSR_OUT(11);
        ALFSR_OUT(13) <= LFSR_OUT(12);
        ALFSR_OUT(14) <= LFSR_OUT(13);
        ALFSR_OUT(15) <= LFSR_OUT(14);
        P_LFSR: process(CLK,EN,RESET,CLRSIG,LFSR_IN,ALFSR_OUT)
        begin
                if (RESET = '0') or (CLRSIG = '1') then             -- asynchronous reset
                        LFSR_OUT <= "0000000000000000";
                elsif CLK = '0' and CLK'event then
                        if EN = '1' then
                                LFSR_OUT <= ALFSR_OUT;      -- latch LFSR output
                        end if;
                end if;
        end process P_LFSR;
end BEHAVIORAL_DESCRIPTION;
```

LFSR (20 bit)

File: lfsr20.vhd

```
entity LFSR20 is
        port(CLK        : in BIT;           -- system clock
        RESET           : in BIT;           -- BIST controller reset
        CLRSIG          : in BIT;           -- clear LFSR signal
        EN              : in BIT;           -- LFSR enable signal
        LFSR_IN         : in BIT;           -- LFSR input, 1 bit wide
        ALFSR           : out BIT_VECTOR(19 downto 0);          -- unlatched LFSR output
        LFSR            : out BIT_VECTOR(19 downto 0));         -- latched LFSR output
end LFSR20;

architecture BEHAVIORAL_DESCRIPTION of LFSR20 is
        signal LFSR_OUT: BIT_VECTOR(19 downto 0);
        signal ALFSR_OUT : BIT_VECTOR(19 downto 0);
begin
        ALFSR <= ALFSR_OUT;
        LFSR <= LFSR_OUT;
        ALFSR_OUT(0) <= LFSR_IN xor LFSR_OUT(19);
        ALFSR_OUT(1) <= LFSR_OUT(0);
        ALFSR_OUT(2) <= LFSR_OUT(1);
        ALFSR_OUT(3) <= LFSR_OUT(2) xor LFSR_OUT(19);
        ALFSR_OUT(4) <= LFSR_OUT(3);
        ALFSR_OUT(5) <= LFSR_OUT(4);
        ALFSR_OUT(6) <= LFSR_OUT(5);
        ALFSR_OUT(7) <= LFSR_OUT(6);
        ALFSR_OUT(8) <= LFSR_OUT(7);
        ALFSR_OUT(9) <= LFSR_OUT(8);
        ALFSR_OUT(10) <= LFSR_OUT(9);
        ALFSR_OUT(11) <= LFSR_OUT(10);
        ALFSR_OUT(12) <= LFSR_OUT(11);
        ALFSR_OUT(13) <= LFSR_OUT(12);
        ALFSR_OUT(14) <= LFSR_OUT(13);
        ALFSR_OUT(15) <= LFSR_OUT(14);
        ALFSR_OUT(16) <= LFSR_OUT(15);
        ALFSR_OUT(17) <= LFSR_OUT(16);
        ALFSR_OUT(18) <= LFSR_OUT(17);
        ALFSR_OUT(19) <= LFSR_OUT(18);
        P_LFSR: process(CLK,EN,RESET,CLRSIG,LFSR_IN,ALFSR_OUT)
        begin
                if (RESET = '0') or (CLRSIG = '1') then                 --asynchronous reset
                        LFSR_OUT <= "00000000000000000000";
                elsif CLK = '0' and CLK'event then
                        if EN = '1' then
                                LFSR_OUT <= ALFSR_OUT;      --latch LFSR output
                        end if;
                end if;
        end process P_LFSR;
end BEHAVIORAL_DESCRIPTION;
```

LFSR (25 bit)

File: lfsr25.vhd

```
entity LFSR25 is
        port(CLK        : in BIT;       -- system clock
        RESET           : in BIT;       -- BIST controller reset
        CLRSIG          : in BIT;       -- clear LFSR signal
        EN              : in BIT;       -- LFSR enable signal
        LFSR_IN         : in BIT;       -- LFSR input, 1 bit wide
        ALFSR           : out BIT_VECTOR(24 downto 0);          -- unlatched LFSR output
        LFSR            : out BIT_VECTOR(24 downto 0));         -- latched LFSR output
end LFSR25;

architecture BEHAVIORAL_DESCRIPTION of LFSR25 is
        signal LFSR_OUT: BIT_VECTOR(24 downto 0);
        signal ALFSR_OUT : BIT_VECTOR(24 downto 0);
begin
        ALFSR <= ALFSR_OUT;
        LFSR <= LFSR_OUT;
        ALFSR_OUT(0) <= LFSR_IN xor LFSR_OUT(24);
        ALFSR_OUT(1) <= LFSR_OUT(0);
        ALFSR_OUT(2) <= LFSR_OUT(1);
        ALFSR_OUT(3) <= LFSR_OUT(2) xor LFSR_OUT(24);
        ALFSR_OUT(4) <= LFSR_OUT(3);
        ALFSR_OUT(5) <= LFSR_OUT(4);
        ALFSR_OUT(6) <= LFSR_OUT(5);
        ALFSR_OUT(7) <= LFSR_OUT(6);
        ALFSR_OUT(8) <= LFSR_OUT(7);
        ALFSR_OUT(9) <= LFSR_OUT(8);
        ALFSR_OUT(10) <= LFSR_OUT(9);
        ALFSR_OUT(11) <= LFSR_OUT(10);
        ALFSR_OUT(12) <= LFSR_OUT(11);
        ALFSR_OUT(13) <= LFSR_OUT(12);
        ALFSR_OUT(14) <= LFSR_OUT(13);
        ALFSR_OUT(15) <= LFSR_OUT(14);
        ALFSR_OUT(16) <= LFSR_OUT(15);
        ALFSR_OUT(17) <= LFSR_OUT(16);
        ALFSR_OUT(18) <= LFSR_OUT(17);
        ALFSR_OUT(19) <= LFSR_OUT(18);
        ALFSR_OUT(20) <= LFSR_OUT(19);
        ALFSR_OUT(21) <= LFSR_OUT(20);
        ALFSR_OUT(22) <= LFSR_OUT(21);
        ALFSR_OUT(23) <= LFSR_OUT(22);
        ALFSR_OUT(24) <= LFSR_OUT(23);

        P_LFSR: process(CLK,EN,RESET,CLRSIG,LFSR_IN,ALFSR_OUT)
        begin
                if (RESET = '0') or (CLRSIG = '1') then                 --asynchronous reset
                        LFSR_OUT <= "0000000000000000000000000";
                elsif CLK = '0' and CLK'event then
                        if EN = '1' then
                                LFSR_OUT <= ALFSR_OUT;      --latch LFSR output
                        end if;
                end if;
        end process P_LFSR;
end BEHAVIORAL_DESCRIPTION;
```

LFSR (30 bit)

File: lfsr30.vhd

```
-- 30-bit LFSR

entity LFSR30 is
        port(CLK        : in BIT;           -- system clock
        RESET           : in BIT;           -- BIST controller reset
        CLRSIG          : in BIT;           -- clear LFSR signal
        EN              : in BIT;           -- LFSR enable signal
        LFSR_IN         : in BIT;           -- LFSR input, 1 bit wide
        ALFSR           : out BIT_VECTOR(29 downto 0);          -- unlatched LFSR output
        LFSR            : out BIT_VECTOR(29 downto 0));         -- latched LFSR output
end LFSR30;

architecture BEHAVIORAL_DESCRIPTION of LFSR30 is
        signal LFSR_OUT: BIT_VECTOR(29 downto 0);
        signal ALFSR_OUT : BIT_VECTOR(29 downto 0);
begin
        ALFSR <= ALFSR_OUT;
        LFSR <= LFSR_OUT;
        ALFSR_OUT(0) <= LFSR_IN xor LFSR_OUT(29);
        ALFSR_OUT(1) <= LFSR_OUT(0) xor LFSR_OUT(29);
        ALFSR_OUT(2) <= LFSR_OUT(1) xor LFSR_OUT(29);
        ALFSR_OUT(3) <= LFSR_OUT(2);
        ALFSR_OUT(4) <= LFSR_OUT(3);
        ALFSR_OUT(5) <= LFSR_OUT(4);
        ALFSR_OUT(6) <= LFSR_OUT(5);
        ALFSR_OUT(7) <= LFSR_OUT(6);
        ALFSR_OUT(8) <= LFSR_OUT(7);
        ALFSR_OUT(9) <= LFSR_OUT(8);
        ALFSR_OUT(10) <= LFSR_OUT(9);
        ALFSR_OUT(11) <= LFSR_OUT(10);
        ALFSR_OUT(12) <= LFSR_OUT(11);
        ALFSR_OUT(13) <= LFSR_OUT(12);
        ALFSR_OUT(14) <= LFSR_OUT(13);
        ALFSR_OUT(15) <= LFSR_OUT(14);
        ALFSR_OUT(16) <= LFSR_OUT(15);
        ALFSR_OUT(17) <= LFSR_OUT(16);
        ALFSR_OUT(18) <= LFSR_OUT(17);
        ALFSR_OUT(19) <= LFSR_OUT(18);
        ALFSR_OUT(20) <= LFSR_OUT(19);
        ALFSR_OUT(21) <= LFSR_OUT(20);
        ALFSR_OUT(22) <= LFSR_OUT(21);
        ALFSR_OUT(23) <= LFSR_OUT(22) xor LFSR_OUT(29);
        ALFSR_OUT(24) <= LFSR_OUT(23);
        ALFSR_OUT(25) <= LFSR_OUT(24);
        ALFSR_OUT(26) <= LFSR_OUT(25);
        ALFSR_OUT(27) <= LFSR_OUT(26);
        ALFSR_OUT(28) <= LFSR_OUT(27);
        ALFSR_OUT(29) <= LFSR_OUT(28);
```

```
        P_LFSR: process(CLK,EN,RESET,CLRSIG,LFSR_IN,ALFSR_OUT)
        begin
                if (RESET = '0') or (CLRSIG = '1') then                 --asynchronous reset
                        LFSR_OUT <= "00000000000000000000000000000000";
                elsif CLK = '0' and CLK'event then
                        if EN = '1' then
                                LFSR_OUT <= ALFSR_OUT;      --latch LFSR output
                        end if;
                end if;
        end process P_LFSR;
end BEHAVIORAL_DESCRIPTION;
```

Read Bit Flipper

File: readflip.vhd

```
-- read bit flipper

use work.BTYPES.all;
entity READ_BIT_FLIPPER is
        port(B          : in BIT;          --from RAM data bus
        STATE :          in STATES;        --current state
        PEQNBAR         : in BIT;          --P=N bar
        Q               : out BIT);        --ouput to LFSR
end READ_BIT_FLIPPER;

architecture BEHAVIORAL_DESCRIPTION of READ_BIT_FLIPPER is
begin
        P_RBF: process(B,STATE,PEQNBAR)
        begin
                if STATE = S3 then
                        Q <= not B;
                elsif STATE = S7 then
                        Q <= PEQNBAR xor B;
                else
                        Q <= B;
                end if;
        end process P_RBF;
end BEHAVIORAL_DESCRIPTION;
```

K-bit Register

File: register.vhd

```
use work.CONVERT_TYPE.all;

entity REGISTERS is
        generic(K:INTEGER);                                     -- number of bits in register
        port( CLK       : in BIT;                               -- system clock
        RESET           : in BIT;                               --BIST controller reset
        REG_IN          : in BIT_VECTOR(K-1 downto 0);          --register input K bits wide
        LOADREG         : in BIT;                               --register load signal
        REG_OUT         : out BIT_VECTOR(K-1 downto 0));        --register output K bits wide
end REGISTERS;

architecture BEHAVIORAL_DESCRIPTION of REGISTERS is
        signal IREG_OUT : BIT_VECTOR(K-1 downto 0);             --internal register output signal
begin
        REG_OUT <= IREG_OUT;
        P_REG: process(CLK,RESET,LOADREG,REG_IN,IREG_OUT)
        begin
                if RESET = '0' then                             -- asynchronous reset
                        IREG_OUT <= INTEGER_TO_BIT_VECTOR(0,K);
                elsif CLK = '0' and CLK'event then
                        if LOADREG = '1' then
                                IREG_OUT <= REG_IN;  --load register
                        end if;
                end if;
        end process P_REG;
end BEHAVIORAL_DESCRIPTION;
```

K-bit Comparator

File: comparator.vhd

```
-- K-bit comparator

entity COMPARATOR is
        generic(K:INTEGER);                                     --number of bits in inputs A & B
        port(A    : in BIT_VECTOR(K-1 downto 0);                --Input A K bit wide
        B         : in BIT_VECTOR(K-1 downto 0);                --Input B K bit wide
        AeqB      : out BIT);                                   --A = B output
end COMPARATOR;
architecture BEHAVIORAL_DESCRIPTION of COMPARATOR is
begin
        P_COMP: process(A,B)
        begin
                if A = B then
                        AeqB <= '1';                            -- AeqB = 1 if A=B
                else
                        AeqB <= '0';
                end if;
        end process P_COMP;
end BEHAVIORAL_DESCRIPTION;
```

Test_Pattern_Generator

File: test_pat.vhd

```
-- test pattern generator

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.CONVERT_TYPE.all;

entity TEST_PATTERN_GENERATOR is
        port( CLK       : in BIT;               --system clock
        RESET           : in BIT;               --BIST controller reset
        WRITE           : in BIT;               --BIST controller write
        READ            : in BIT;               --BIST controller read
        B               : in BIT;               --from RAM data
        Q               : out X01Z);            --data to be written to RAM
end TEST_PATTERN_GENERATOR;

architecture BEHAVIORAL_DESCRIPTION of TEST_PATTERN_GENERATOR is
        signal R : BIT;                         -- D flip-flop output
begin
P_TPG : process(CLK,RESET,READ,WRITE,B,R)
        begin
                if RESET = '0' then R <= '0';
                elsif CLK = '0'and CLK'event then
                        if READ = '1' then
                                R <= not(B);    --load D flip-flop with B bar
                        end if;
                end if;
                if WRITE = '1' then
                        Q <= CONVERT_TO_X01Z(R);--put R into data bus
                else
                        Q <= 'Z';               --isolate data bus
                end if;
        end process P_TPG;
end BEHAVIORAL_DESCRIPTION;
```

PHASE2 Latch

File: phase2latch.vhd

```
--PHASE2 latch

entity PHASE2LATCH is
        port(CLR        : in BIT;
        CLRPHASE2       : in BIT;
        EN              : in BIT;
        D               : in BIT;
        Q               : out BIT);
end PHASE2LATCH;

architecture BEHAVIORAL_DESCRIPTION of PHASE2LATCH is
begin
P_LATCH:process(CLR,EN,D,CLRPHASE2)
begin
        if CLR = '0' or CLRPHASE2 = '1' then Q <= '0';      --asynchronous reset
        elsif EN = '1' then  Q <= '1';                      --asynchronous set
        else Q <= D;
        end if;
end process P_LATCH;
end BEHAVIORAL_DESCRIPTION;
```

PASS / $\overline{\text{FAIL}}$ Latch

File: passlatch.vhd

```
-- PASS/FAIL latch

entity PASSLATCH is
        port(CLR        : in BIT;
        EN              : in BIT;
        D               : in BIT;
        Q               : out BIT);
end PASSLATCH;

architecture BEHAVIORAL_DESCRIPTION of PASSLATCH is
begin
P_LATCH:process(CLR,EN,D)
begin
        if CLR = '1'  then Q <= '0';                        --asynchronous reset
        elsif EN = '0' then  Q <= '1';                      --asynchronous set
        else Q <= D;
        end if;
end process P_LATCH;
end BEHAVIORAL_DESCRIPTION;
```

B Schematics

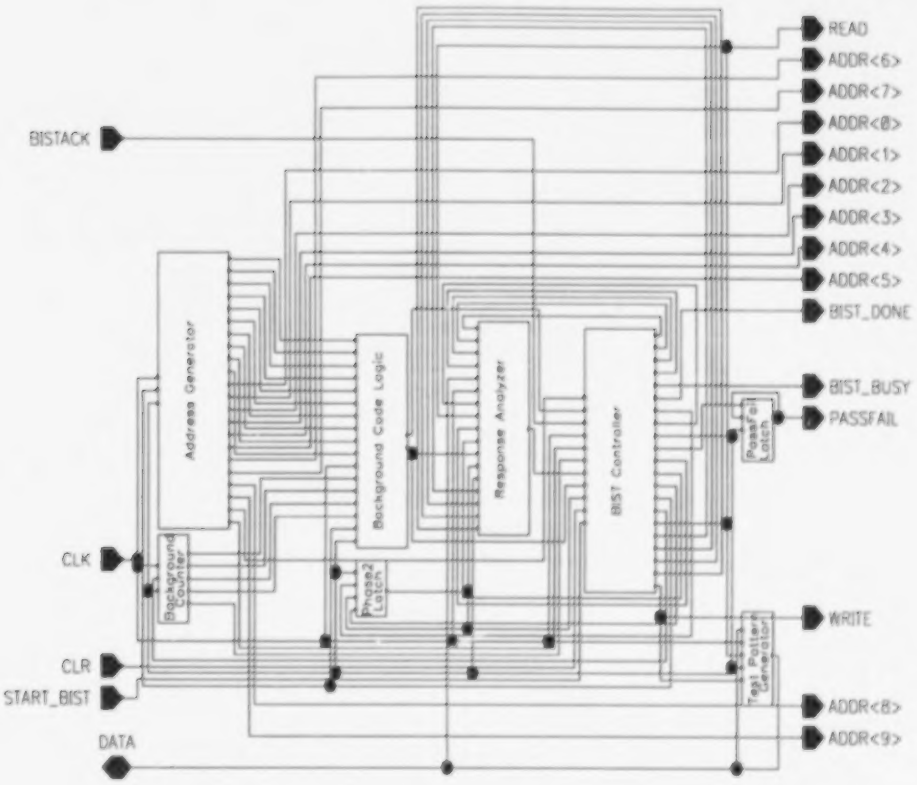


Figure 61: BIST Circuit ($n = 1\text{k}$, $V = 3$)

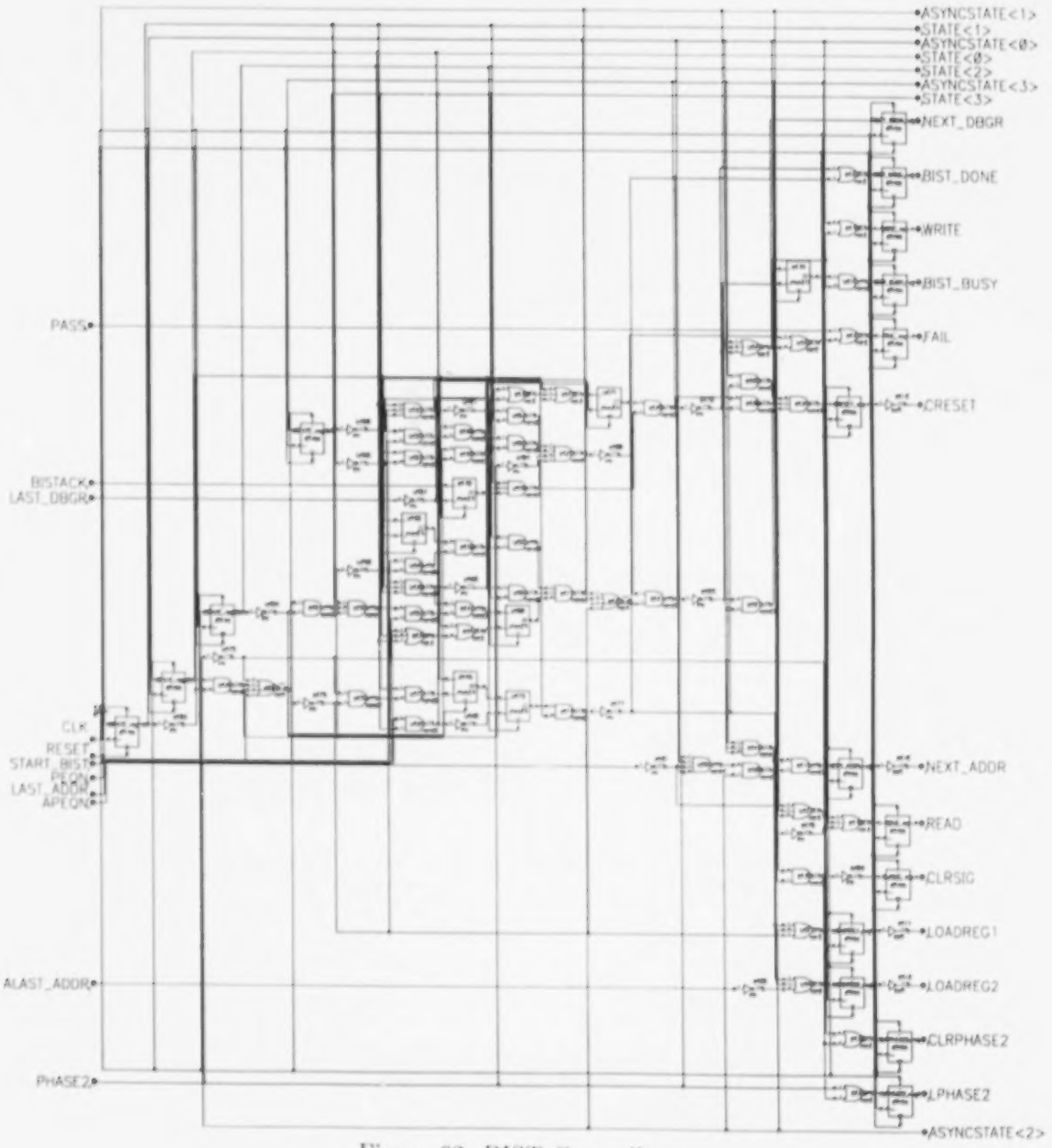


Figure 62: BIST Controller

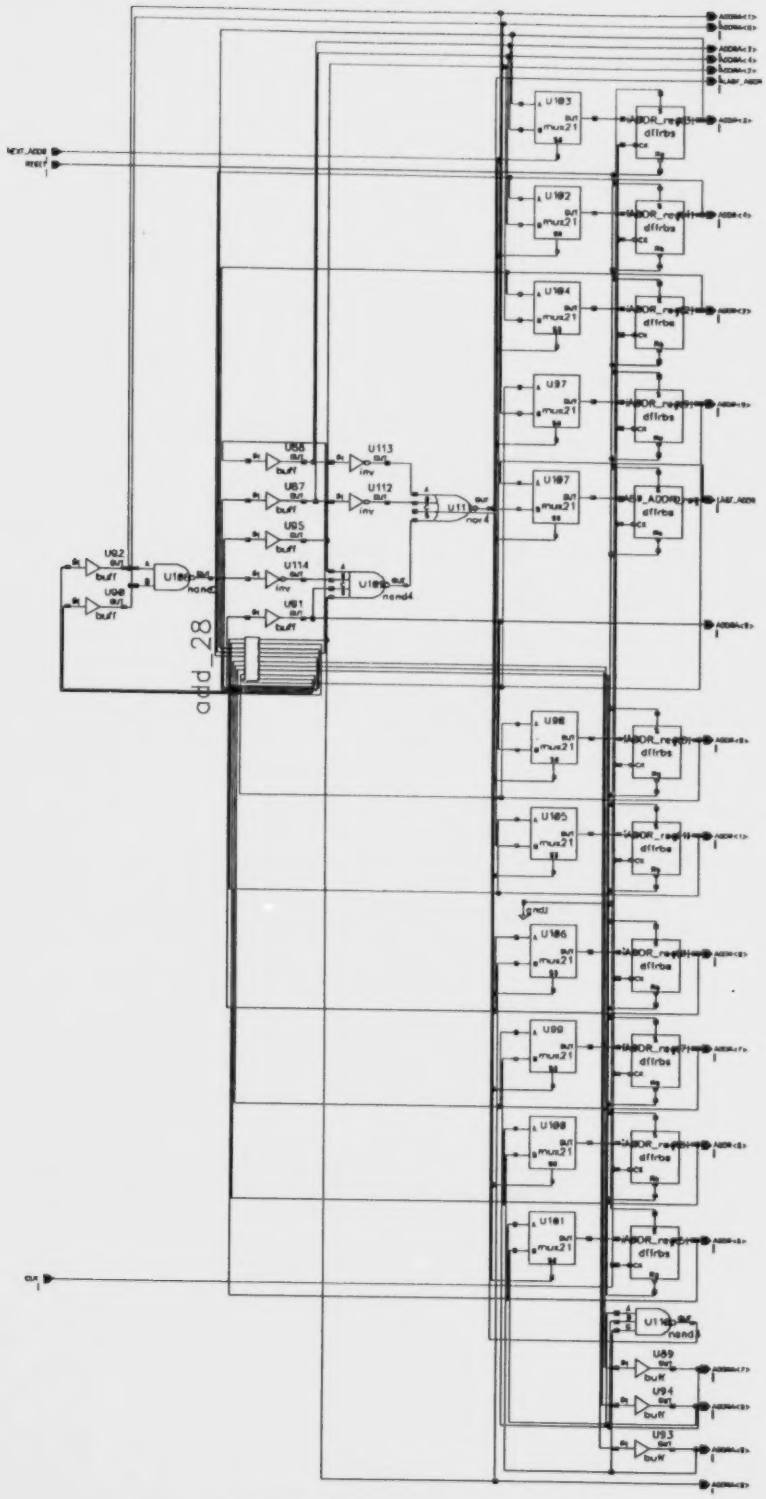


Figure 63: Address Generator

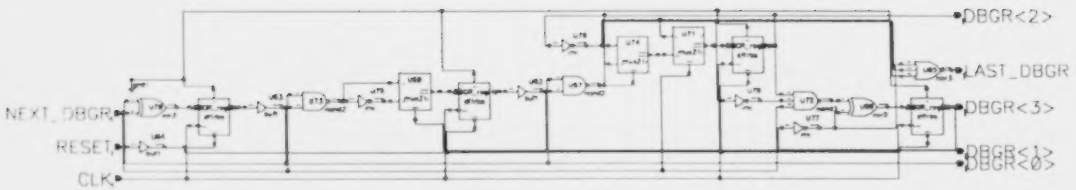


Figure 64: Background Counter ($V = 3$)

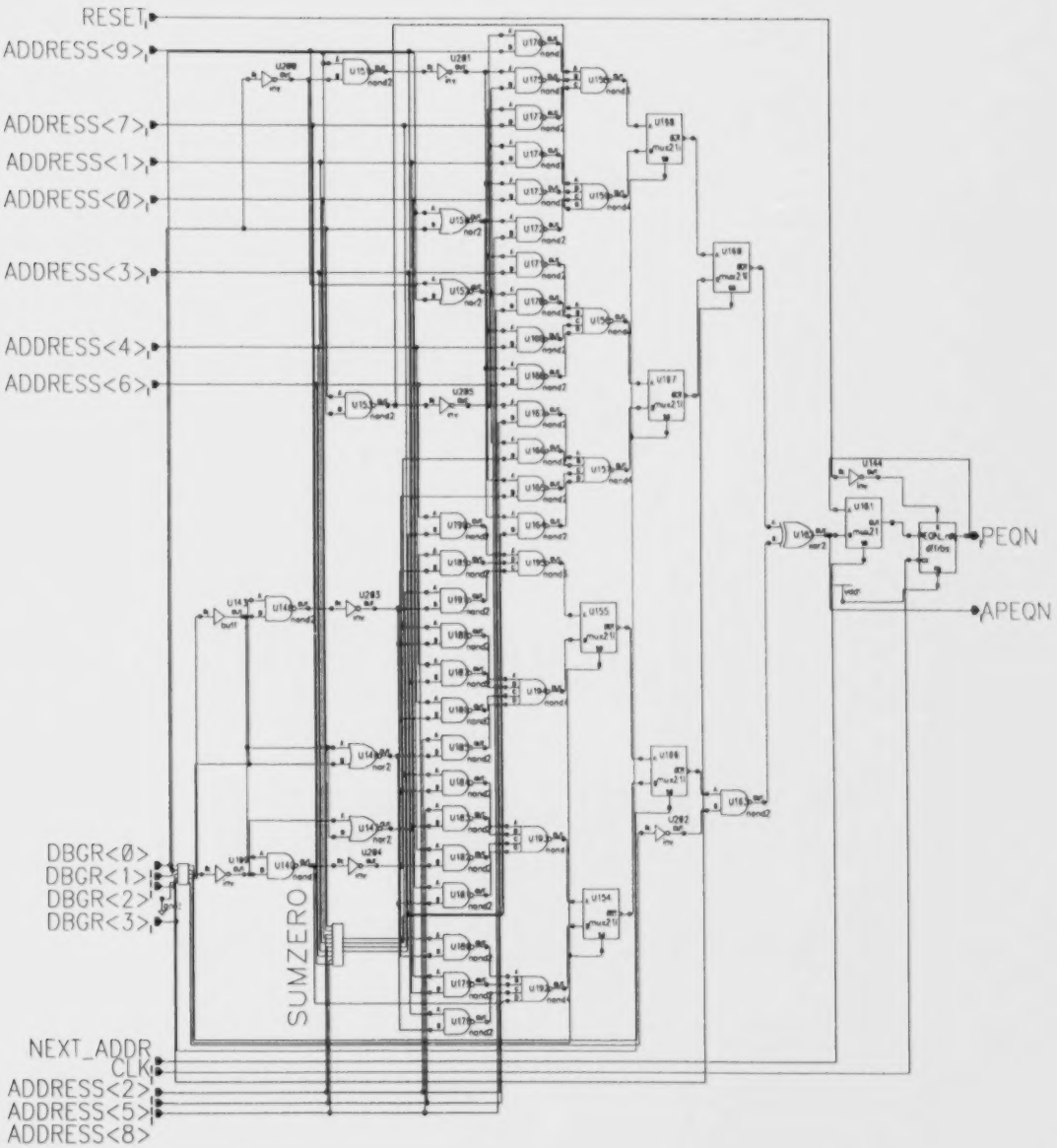


Figure 65: Background Code Logic ($V = 3$)

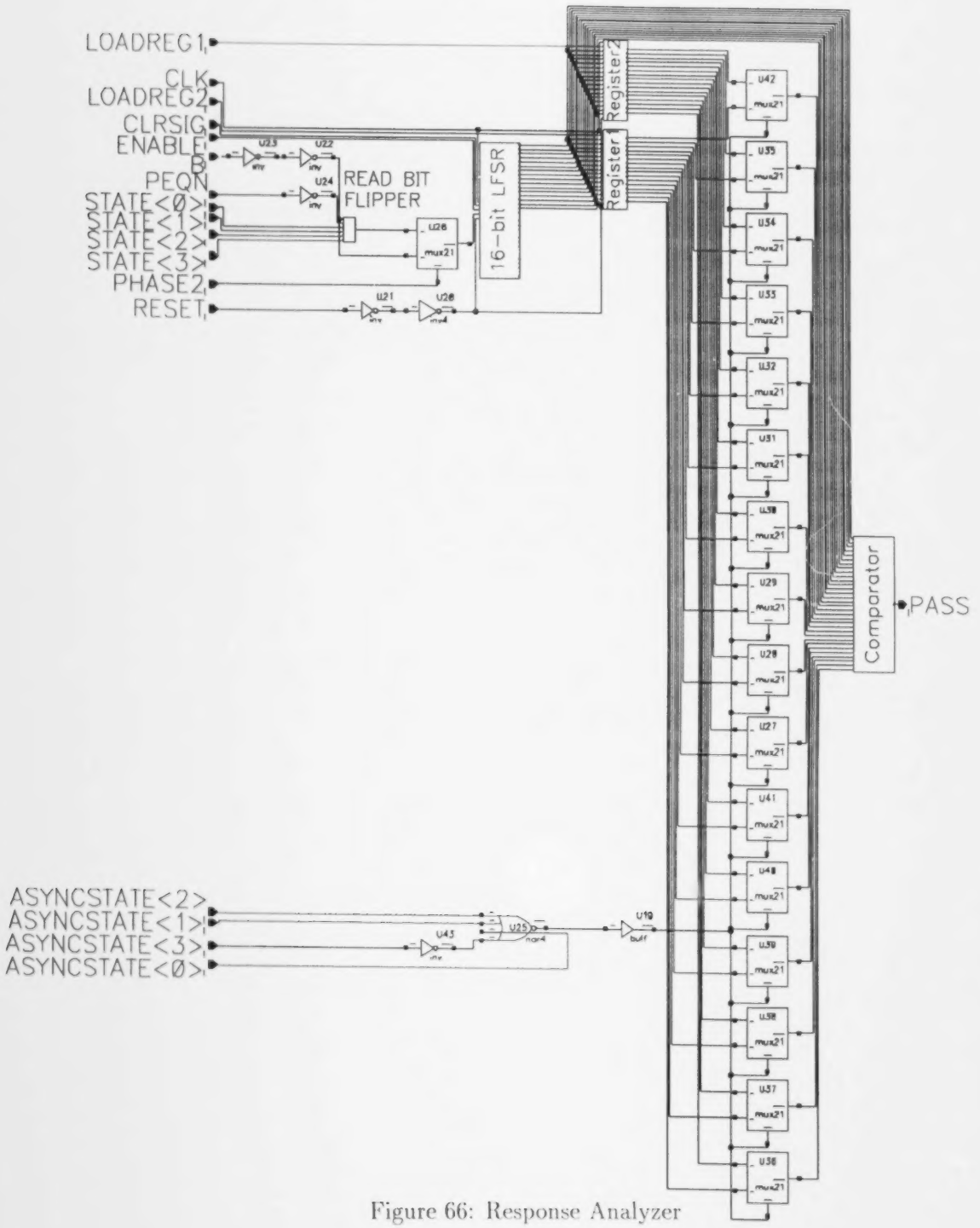


Figure 66: Response Analyzer

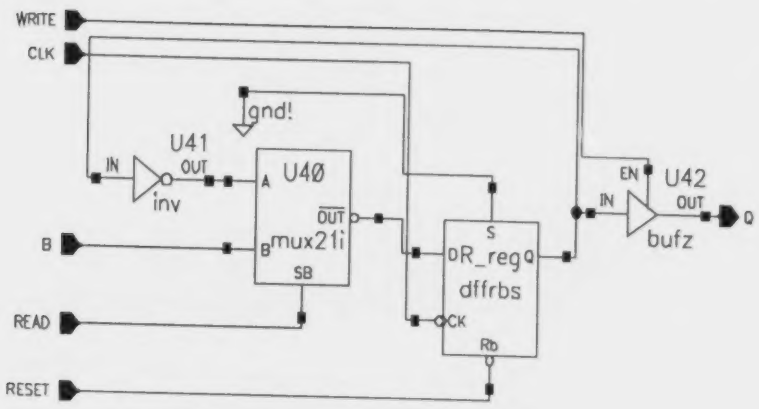


Figure 67: Test Pattern Generator

C Simulation Waveforms

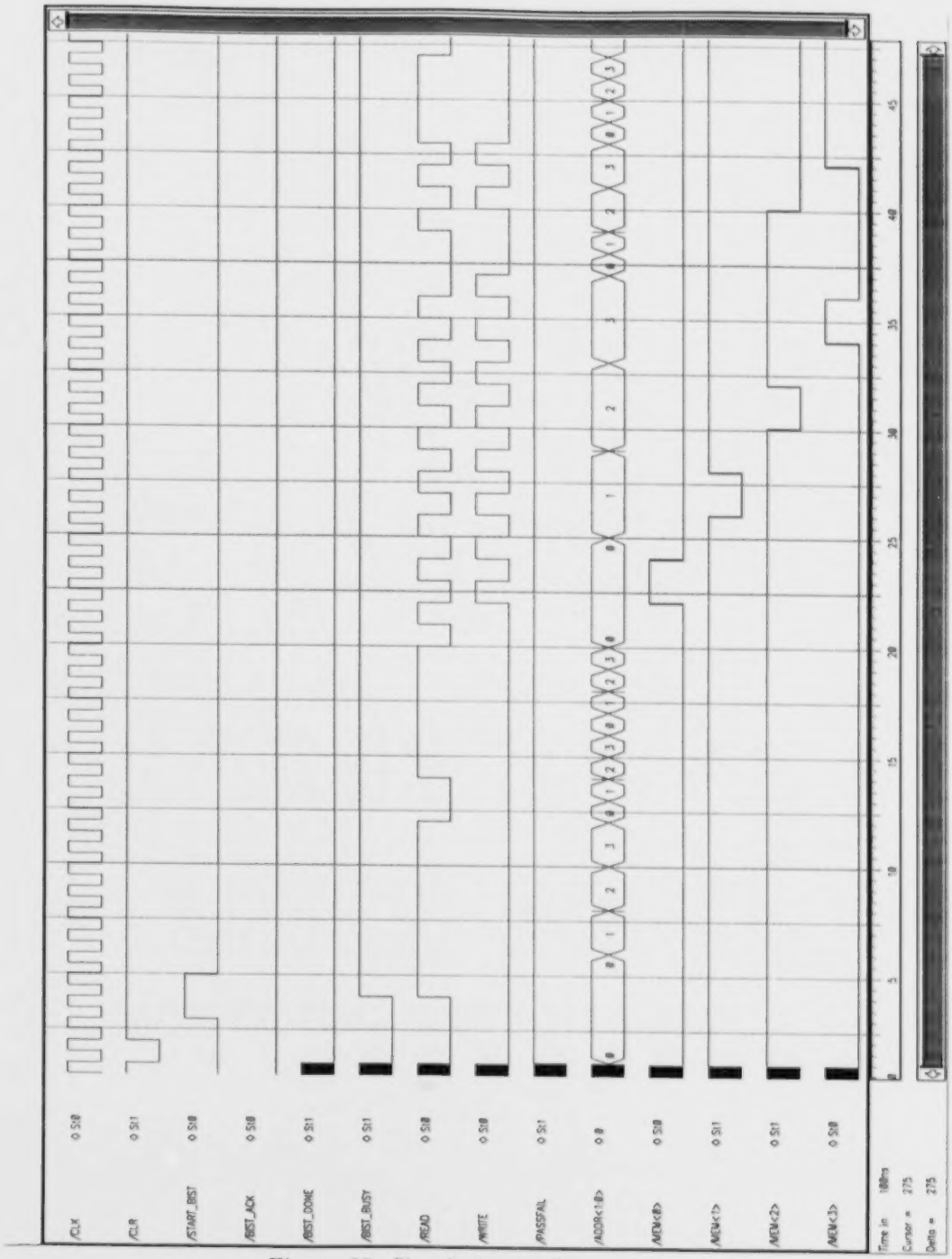


Figure 68: Simulation 1: Segment 1

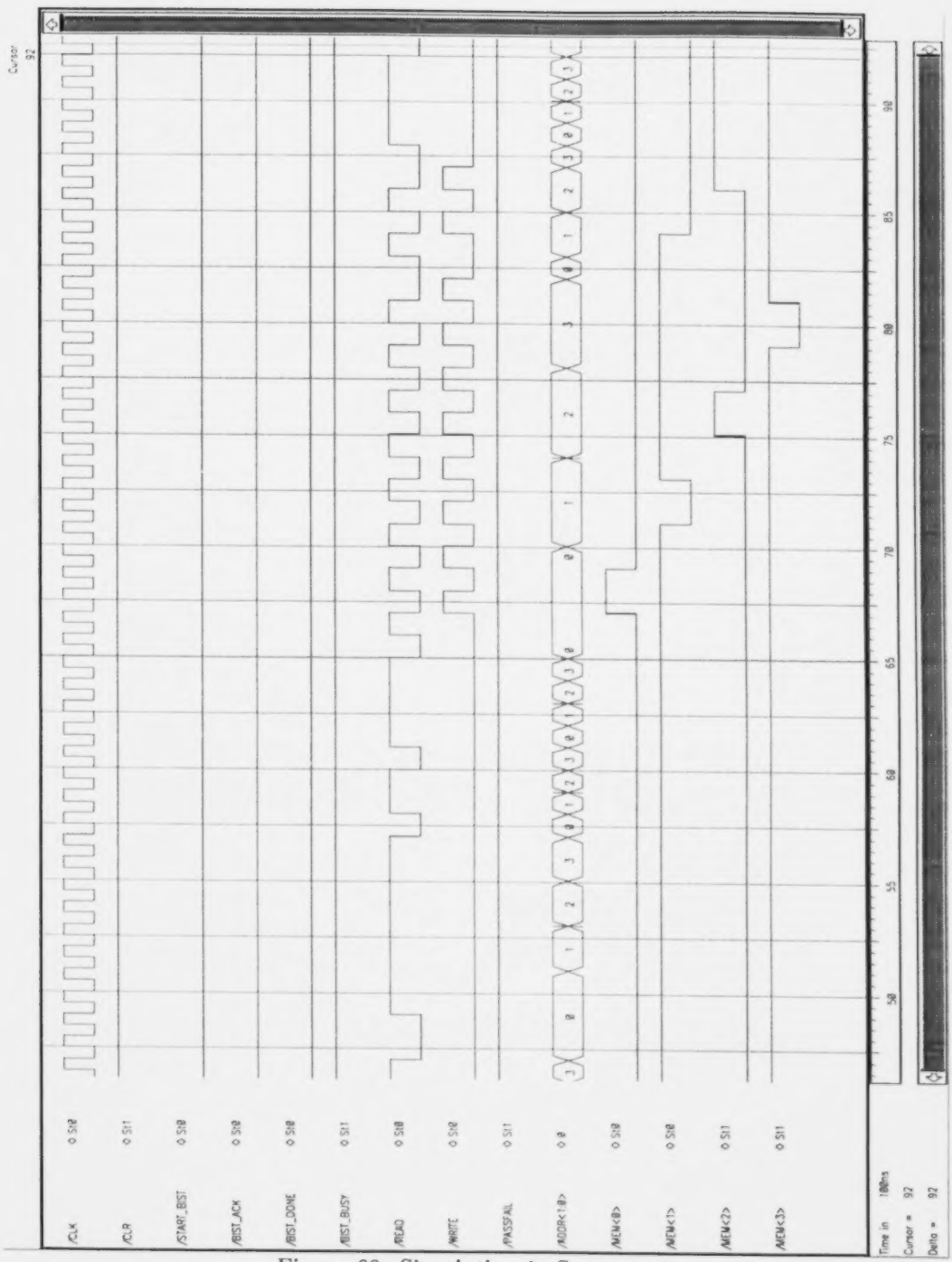


Figure 69: Simulation 1: Segment 2

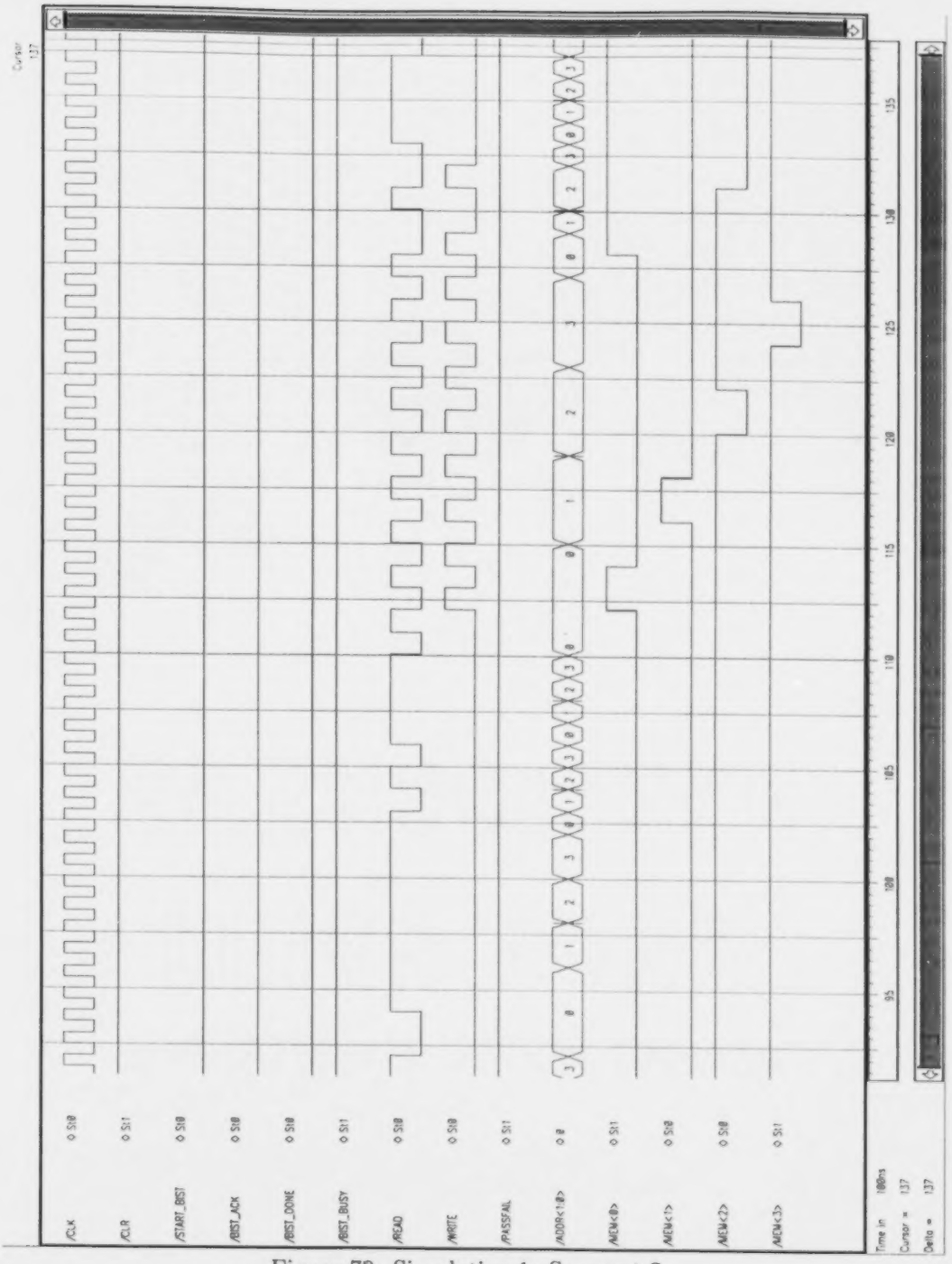


Figure 70: Simulation 1: Segment 3

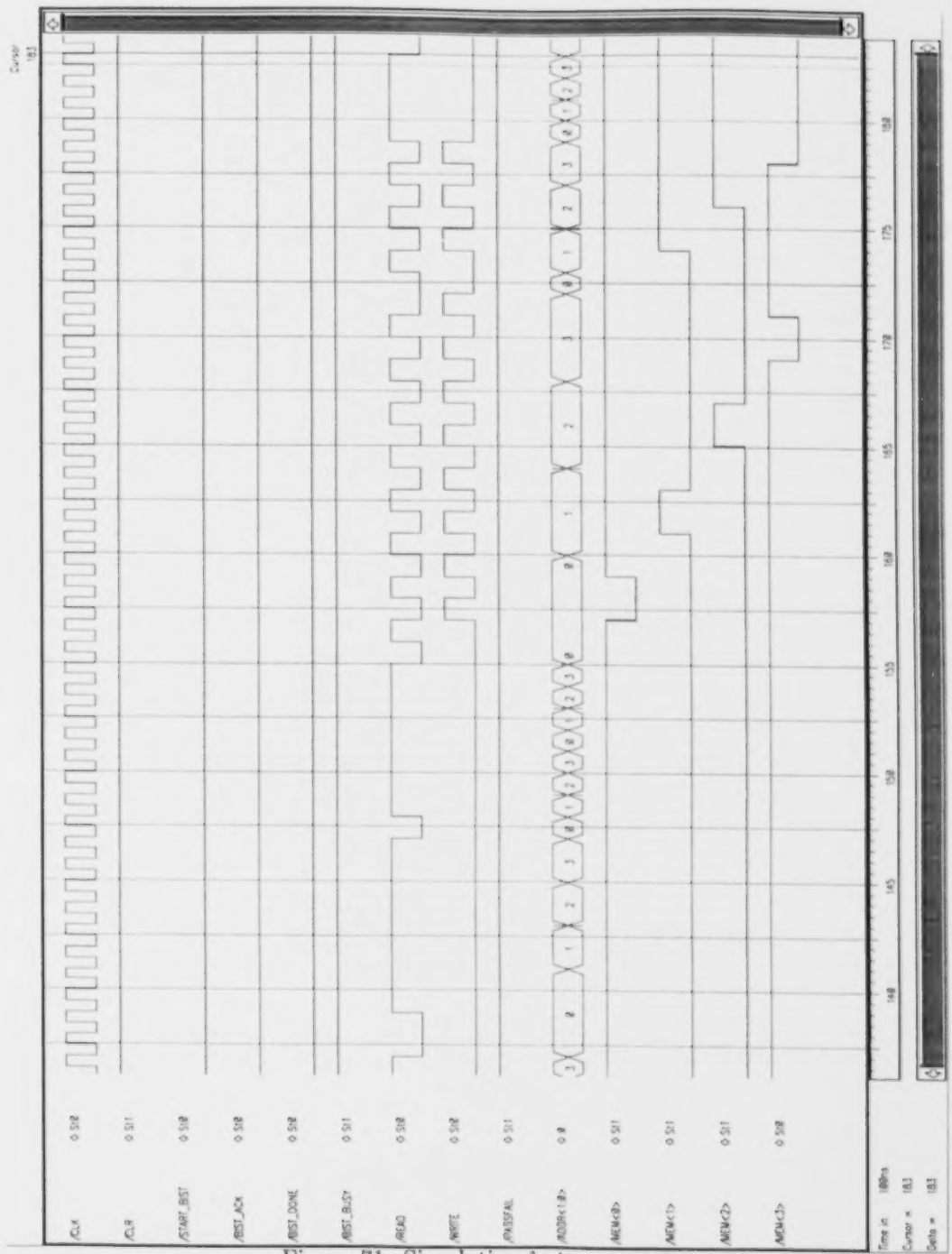


Figure 71: Simulation 1: Segment 4

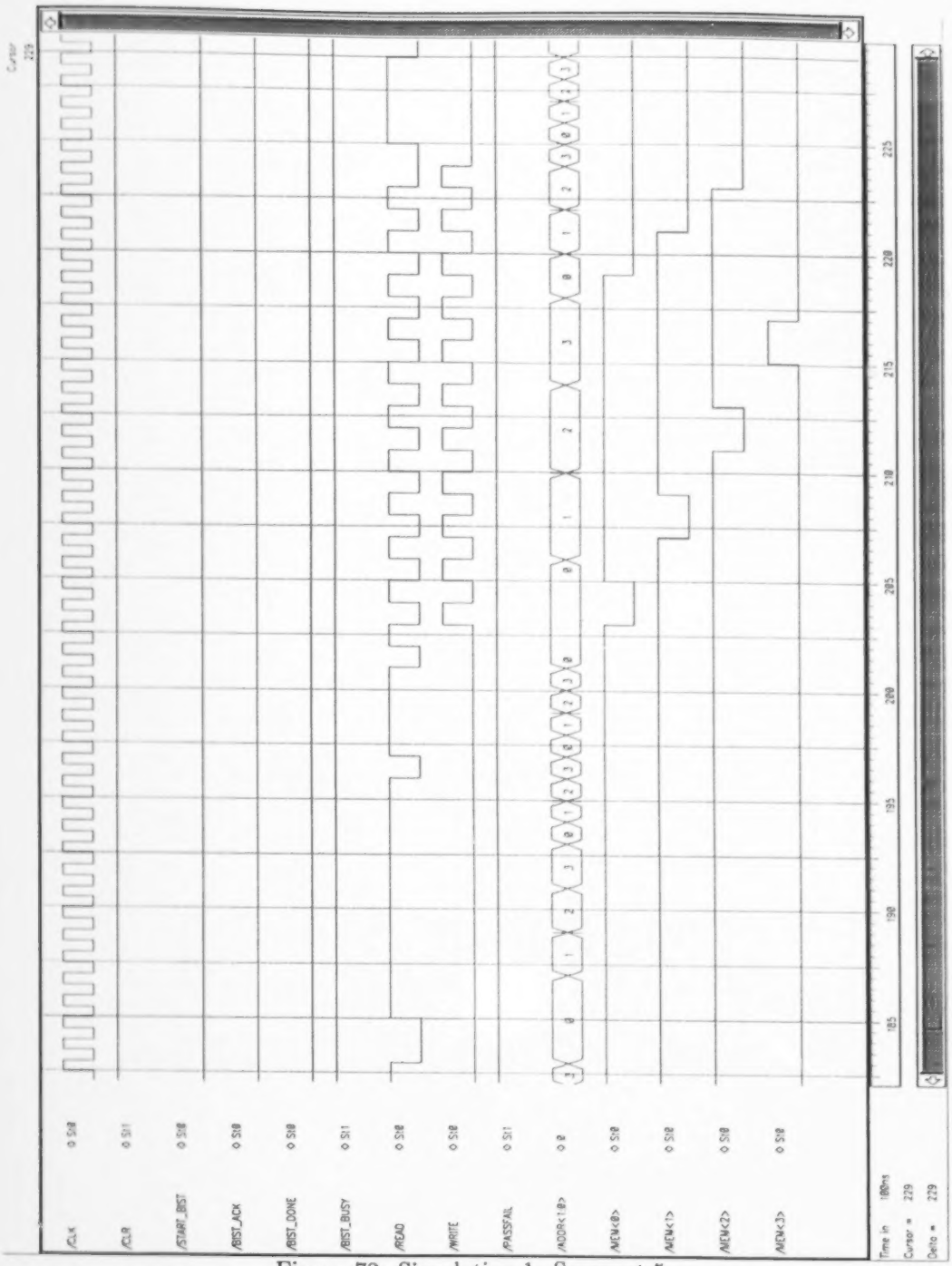


Figure 72: Simulation 1: Segment 5

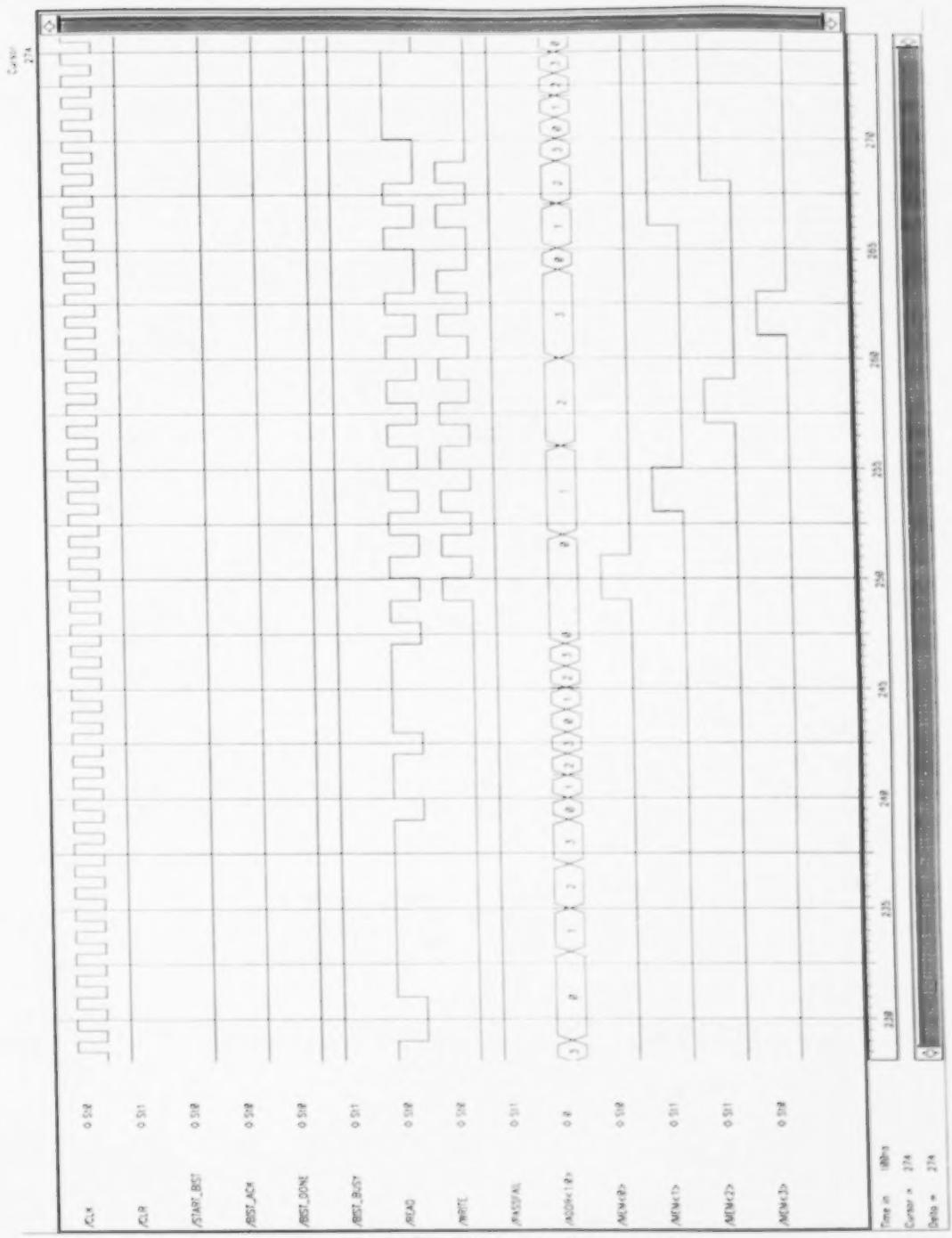


Figure 73: Simulation 1: Segment 6

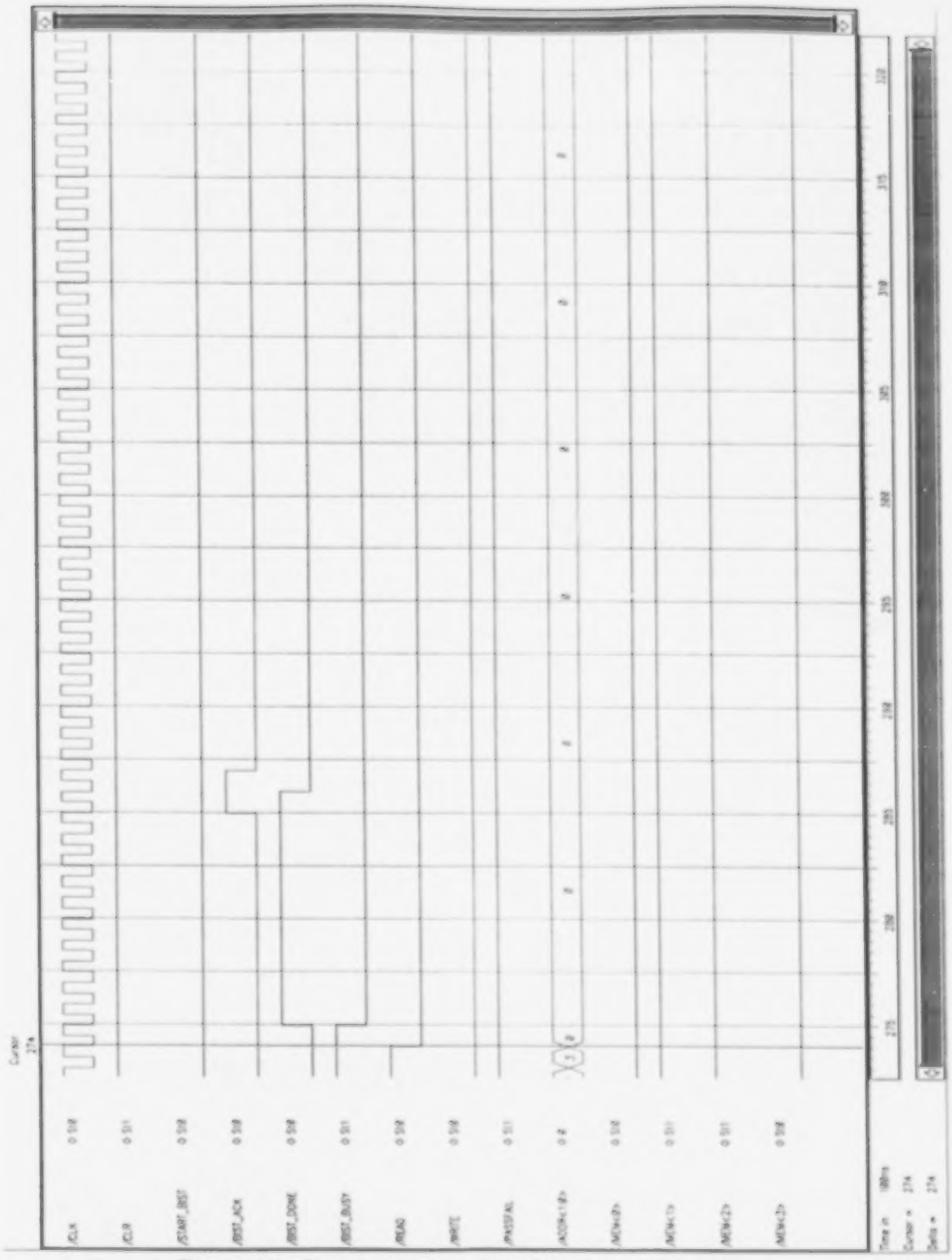


Figure 74: Simulation 1: $t = 27500$ns to $t = 32000$ns

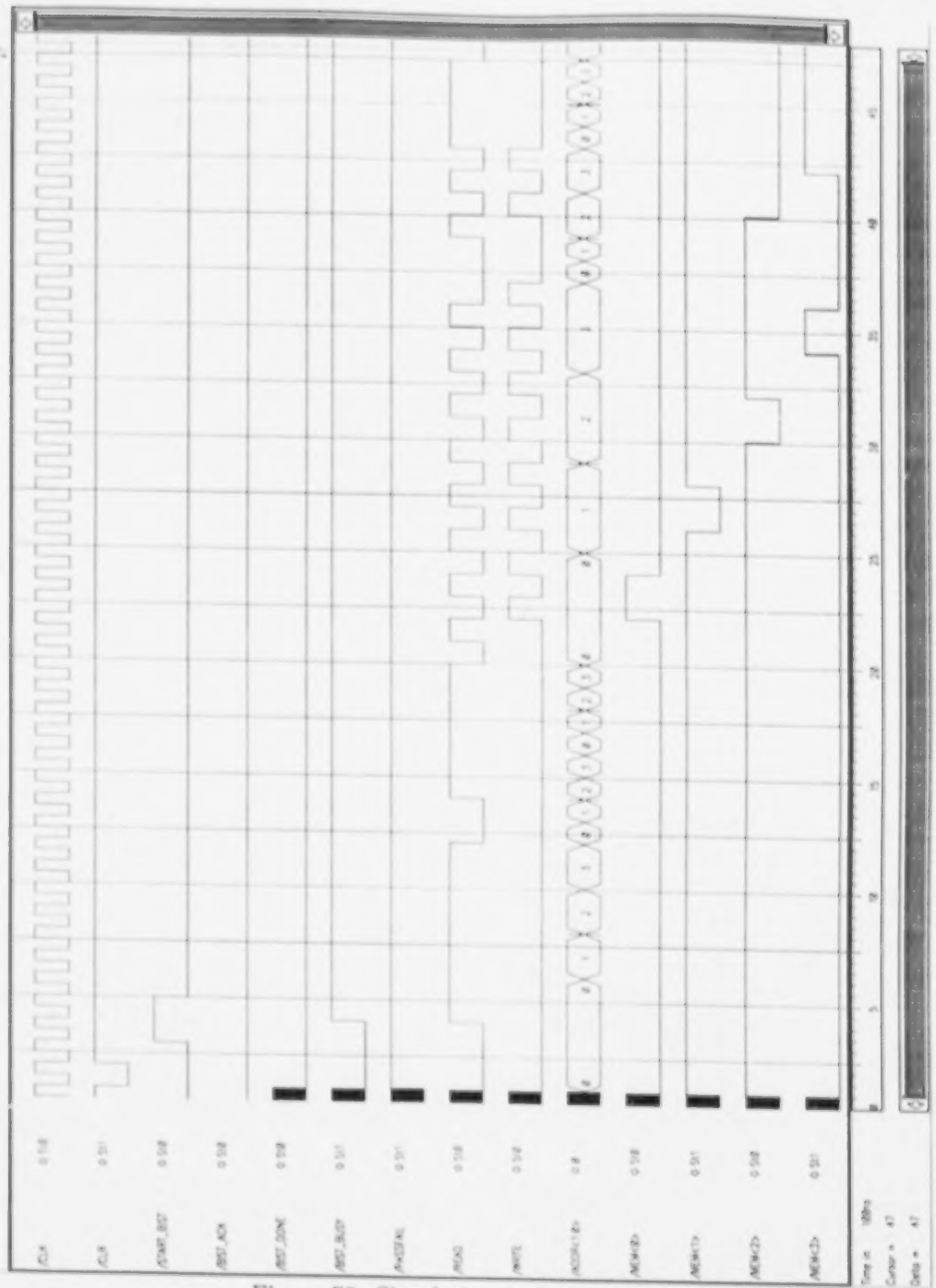


Figure 75: Simulation 2: Segment 1

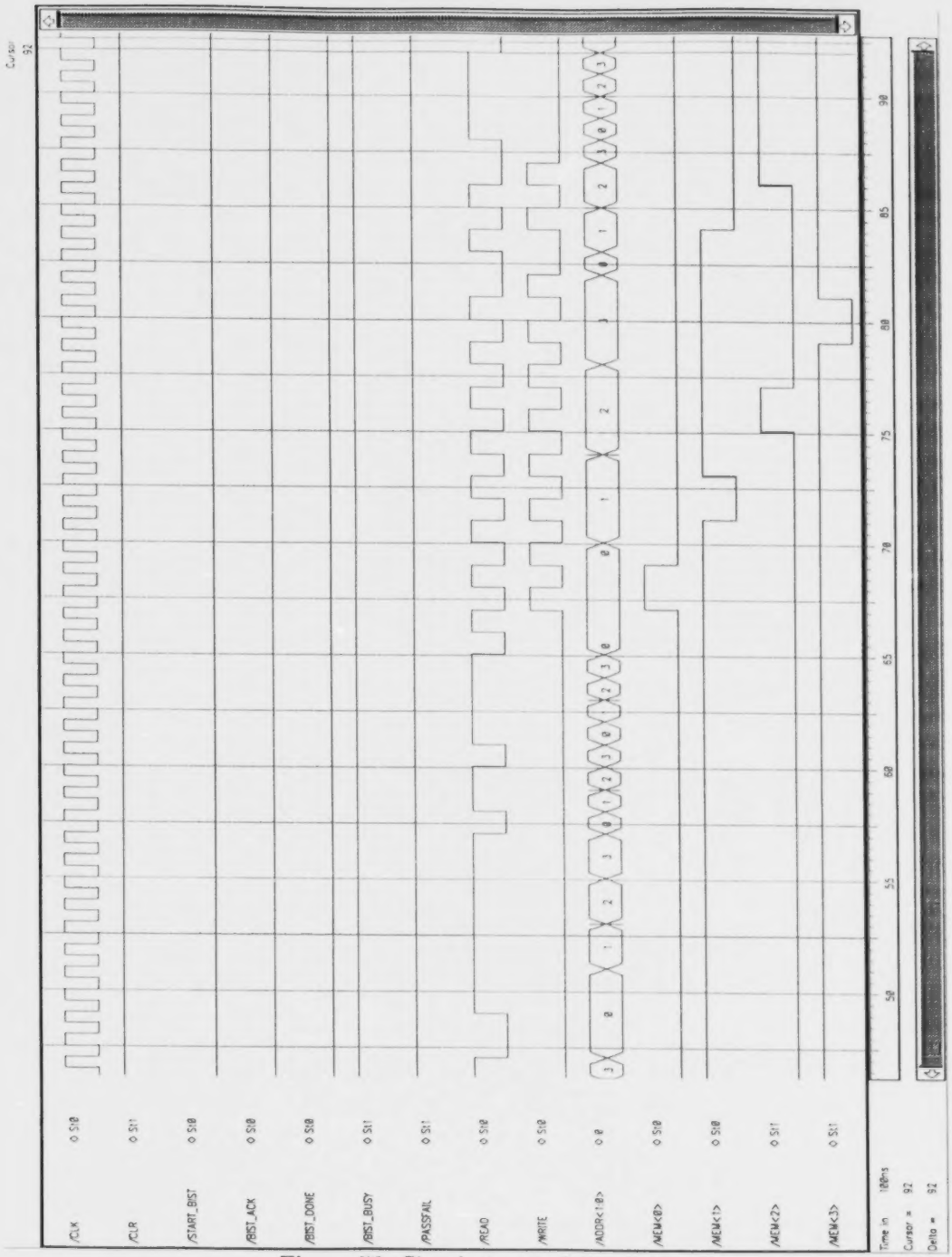


Figure 76: Simulation 2: Segment 2

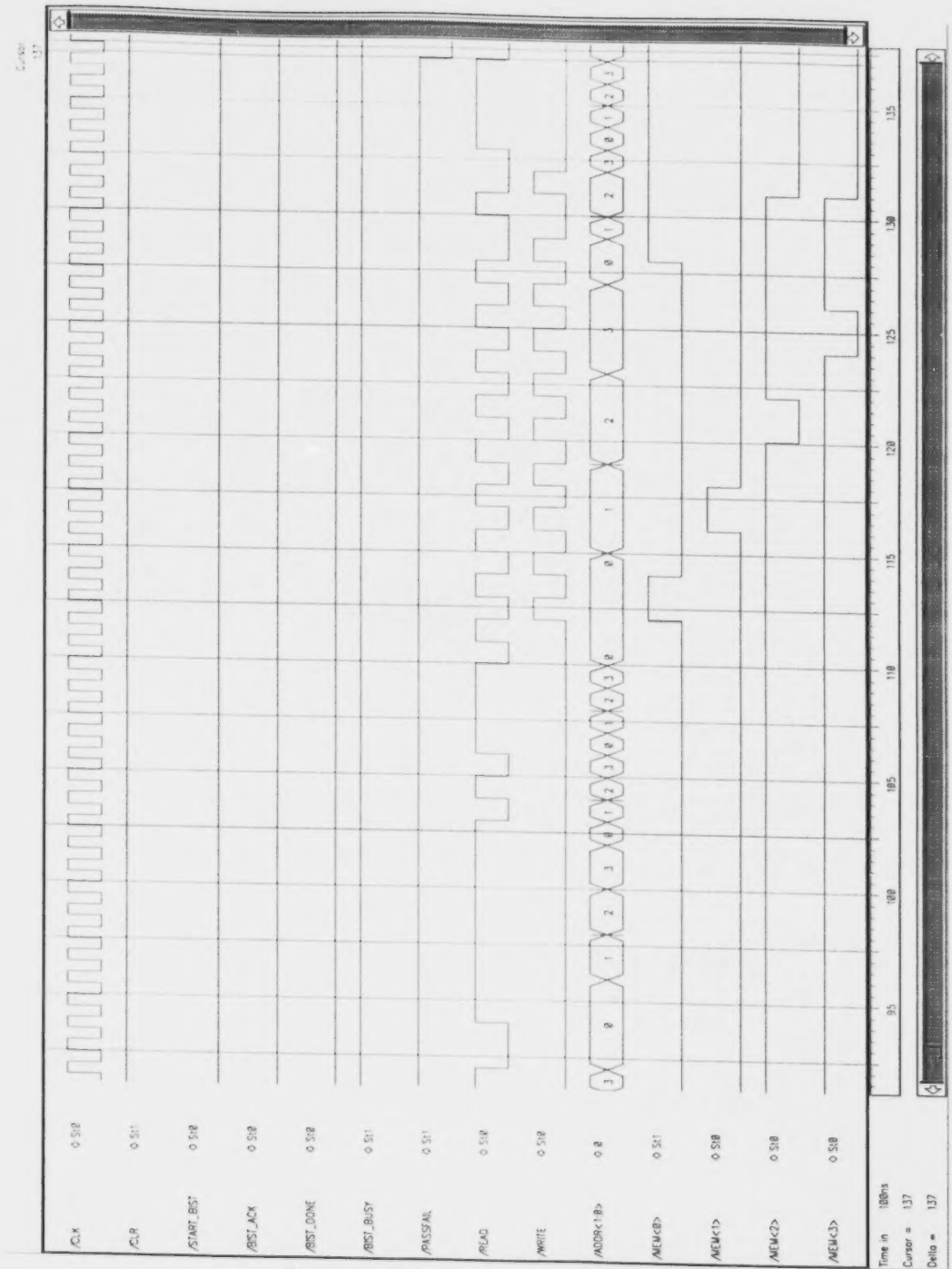


Figure 77: Simulation 2: Segment 3

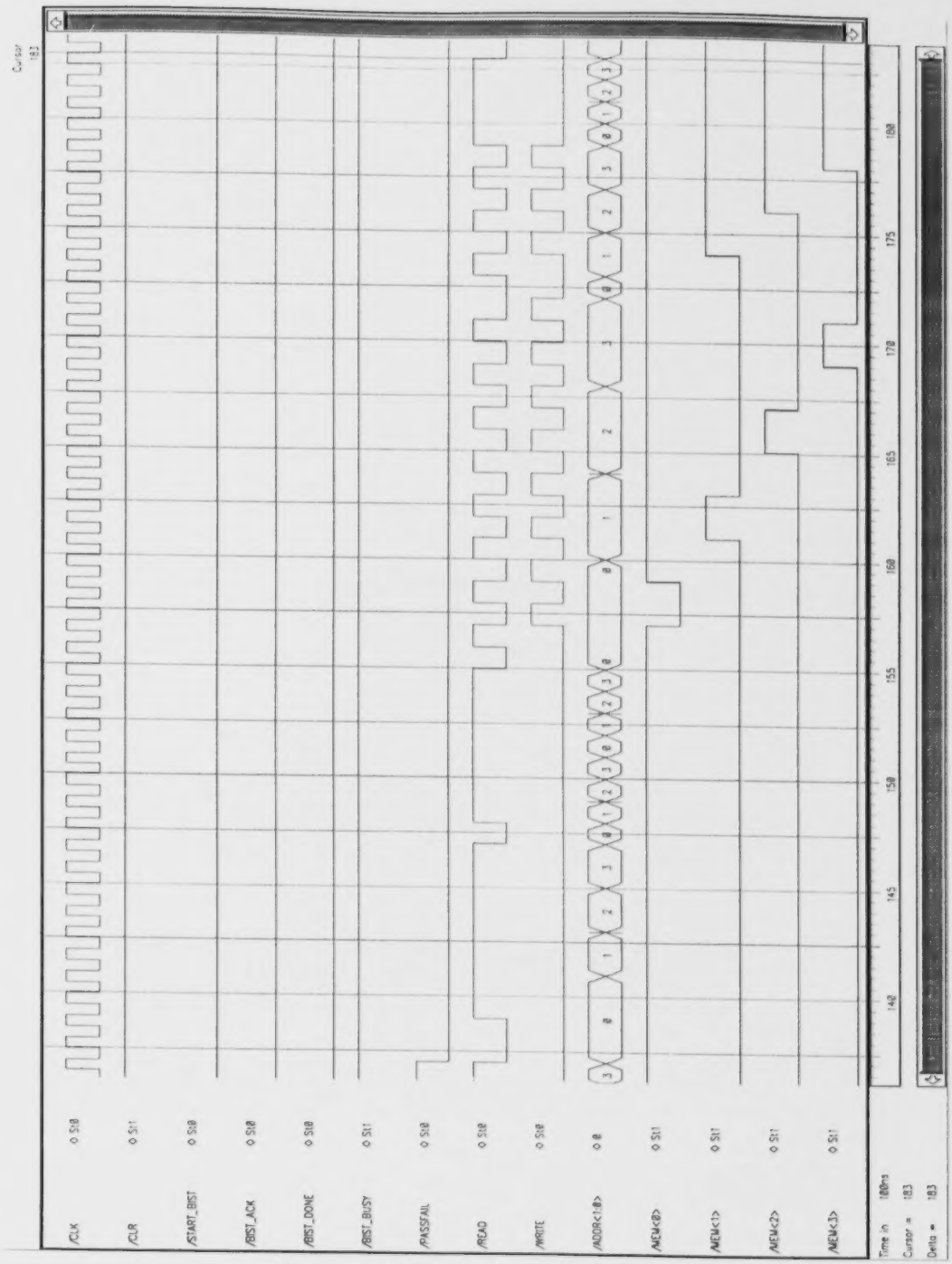


Figure 78: Simulation 2: Segment 4

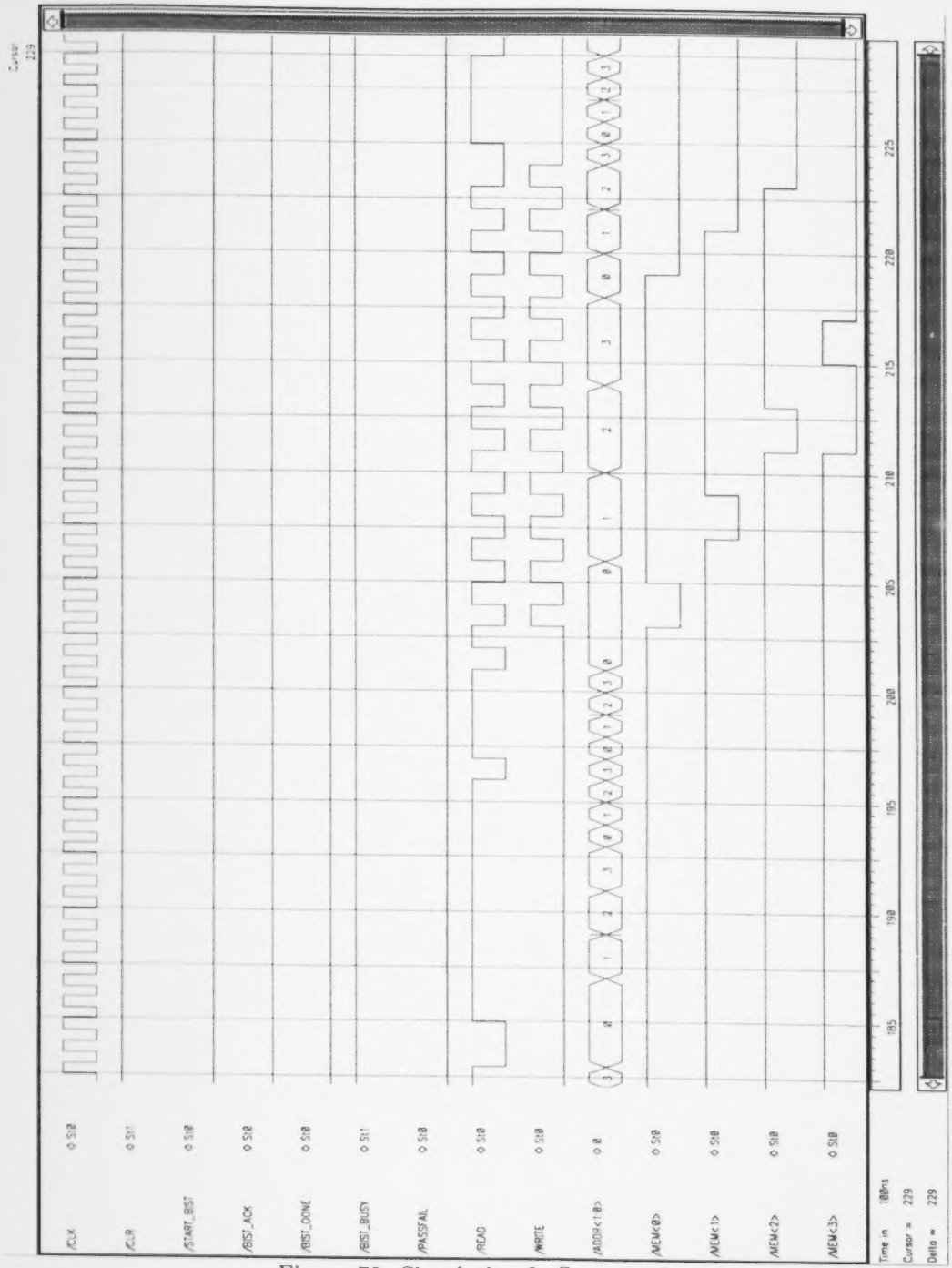


Figure 79: Simulation 2: Segment 5

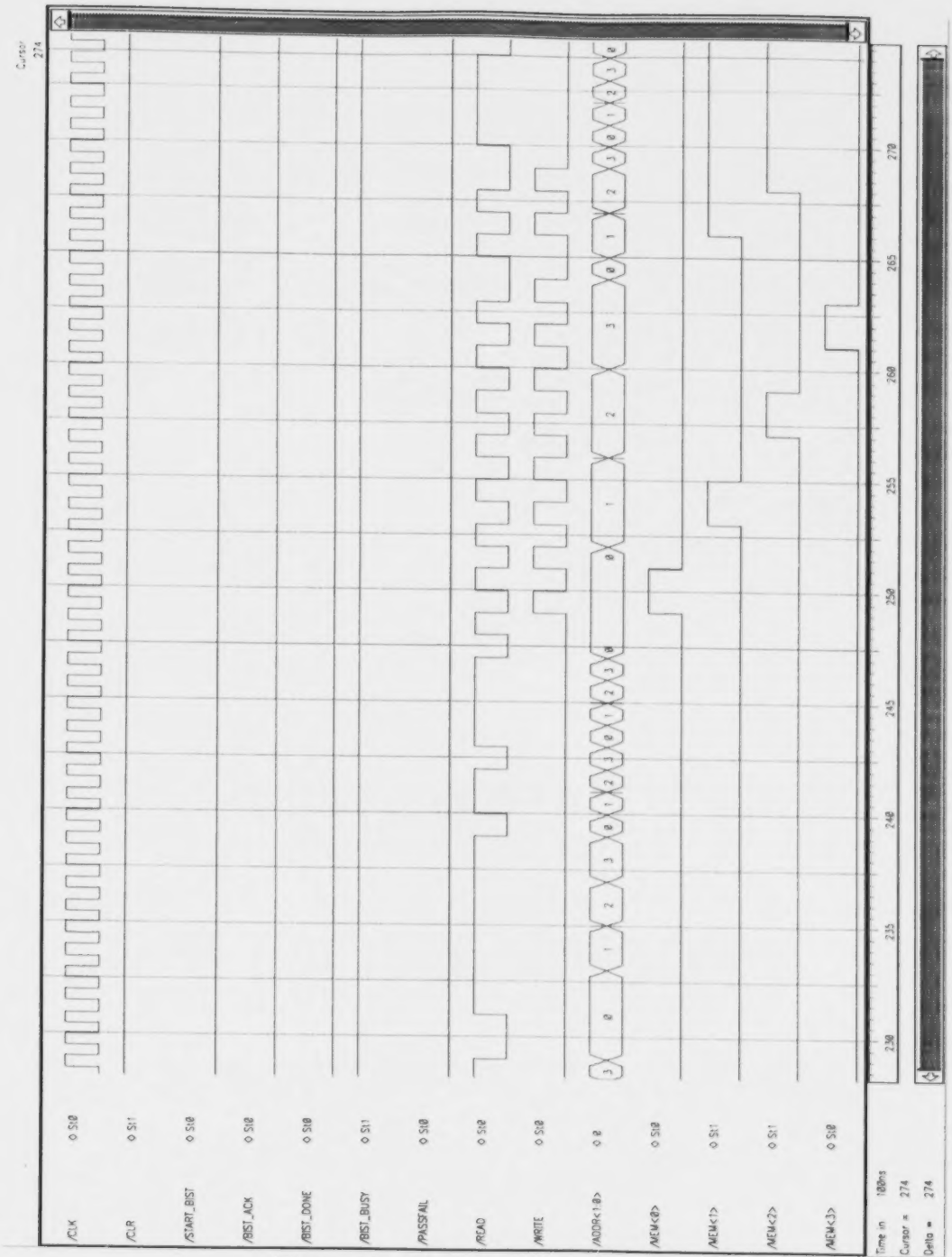


Figure 80: Simulation 2: Segment 6

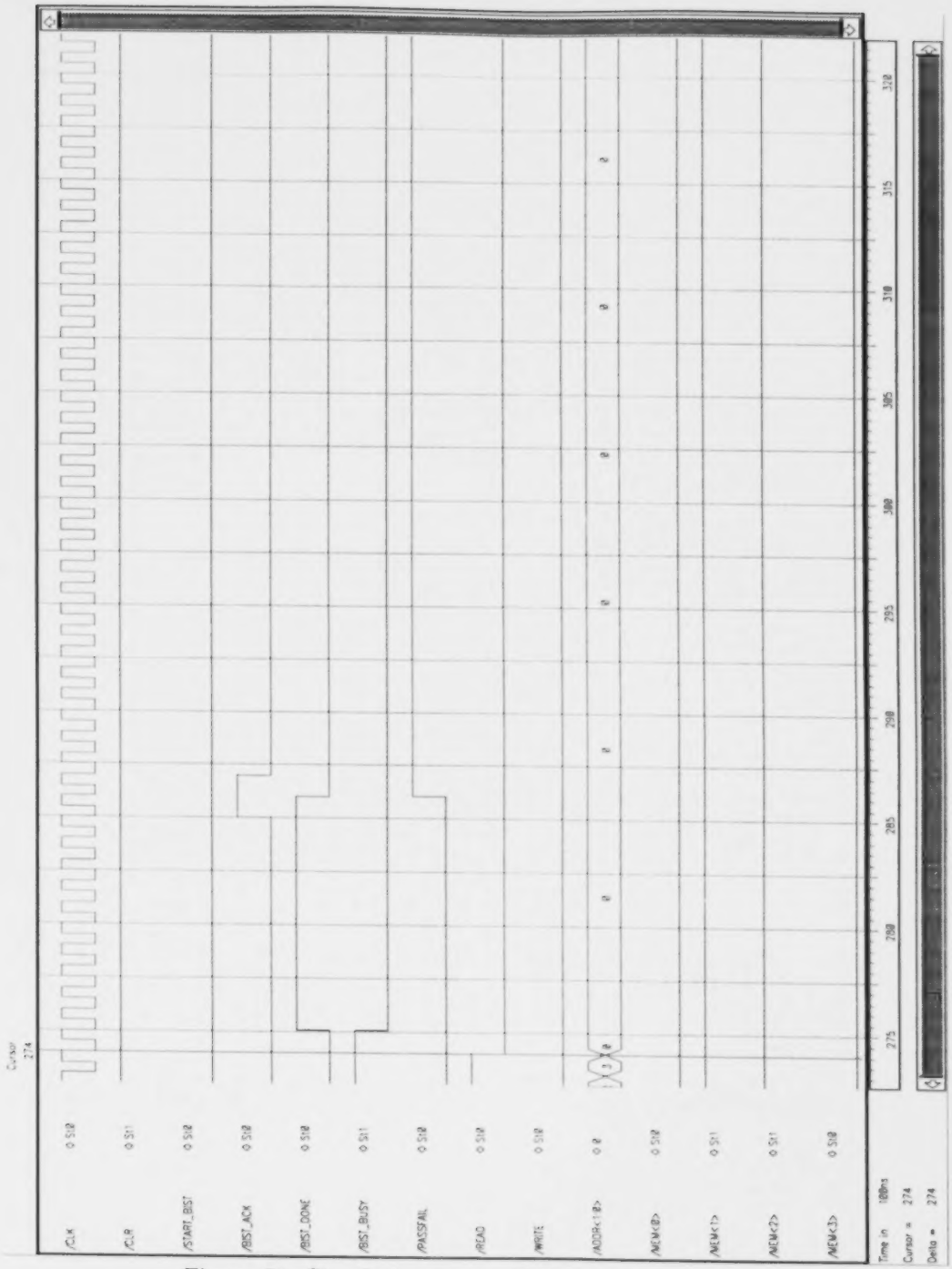


Figure 81: Simulation 2: $t = 27500$ns to $t = 32000$ns